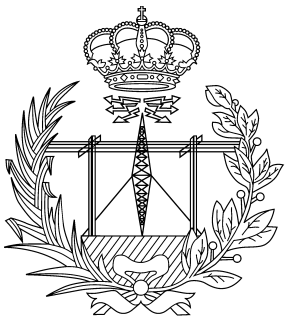
ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INDUSTRIALES Y DE TELECOMUNICACIÓN

UNIVERSIDAD DE CANTABRIA



Trabajo Fin de Máster

ARQUITECTURA FOG/CLOUD EN EL ÁMBITO DEL IIOT: MODELADO, DESPLIEGUE Y ANÁLISIS

(Fog/Cloud Architecture for IIoT scenarios: modeling, deployment, and analysis)

Para acceder al Título de

Máster Universitario en Ingeniería de Telecomunicación

Autor: David Cruz Trueba

Septiembre- 2021

UNIVERSIDAD DE CANTABRIA

E.T.S. DE INGENIEROS INDUSTRIALES Y DE TELECOMUNICACIÓN

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

CALIFICACIÓN DEL TRABAJO FIN DE MÁSTER

Realizado por: David Cruz Trueba
Directores del TFM: Luis Francisco Díez Fernández y Ramón Agüero Calvo
Título: "ARQUITECTURA FOG/CLOUD EN EL ÁMBITO DEL IIOT: MODELADO, DESPLIEGUE Y ANÁLISIS"
Title: "Fog/Cloud Architecture for IIoT scenarios: modeling, deployment, and analysis"

Presentado a examen el día: 21 de septiembre de 2021

para acceder al Título de

MÁSTER UNIVERSITARIO EN INGENIERÍA DE TELECOMUNICACIÓN

Composición del tribunal:
Presidente (Apellidos, Nombre): de la Fuente Rodríguez, Luisa
Secretario (Apellidos, Nombre): Lanza Calderón, Jorge
Vocal (Apellidos, Nombre): Fanjul Vélez, Félix

Este Tribunal ha resuelto otorgar la calificación de: .....

Fdo.: El Presidente	Fdo.: El Secretario
Fdo.: El Vocal	Fdo.: El Director del TFM (solo si es distinto del Secretario)
V^{o} B^{o} del Subdirector	Trabajo Fin de Grado N^{o} (a asignar por Secretaría)

Agradecimientos

En primer lugar, me gustaría agradecer a Luis Francisco Díez y Ramón Agüero, directores de este trabajo, su ayuda en todo momento, sin la cual no hubiese sido posible la realización del mismo. Gracias por vuestra confianza y todo el aprendizaje que me habéis aportado.

También agradecer a mis amigos, por todos los buenos momentos que me han ayudado a desconectar en situaciones complicadas.

Por último, agradezco a mi familia el apoyo recibido, no solo durante el desarrollo de este trabajo, si no durante todo mi recorrido por la universidad. Gracias por aconsejarme y apoyarme en todas las decisiones que he tomado en este periodo.

Resumen

El principal objetivo de este trabajo es el desarrollo de una plataforma de pruebas para arquitecturas *Fog/Cloud* sobre la que desplegar diferentes servicios en el ámbito del IIoT. Se utilizarán técnicas de virtualización para desplegar diferentes nodos, que se conectarán a través de enlaces virtuales, cuyo comportamiento se modificará gracias a la utilización del entorno Mahimahi, que permite modular sus parámetros de comportamiento, para emular las características de las conexiones sobre redes reales.

Posteriormente se modelará la llegada de servicios a los nodos *Fog* (más cercanos a los dispositivos) y se estudiarán diferentes estrategias para repartir su procesado entre los dos niveles de la arquitectura. Se estudiará el comportamiento de diferentes alternativas, analizando el coste correspondiente, así como el retardo total, teniendo en cuenta diferentes características para los nodos considerados: capacidad de procesado, coste, etc.

Abstract

The main purpose of this project is the development of an evaluation platform for Fog/Cloud architectures, where different services in the IIoT field can be deployed. Virtualization techniques will be used to deploy different nodes, which will be connected through virtual links. The behavior of these links will be modified using the Mahimahi environment, which allows to modulate their characteristics, to emulate the behavior of real network connections.

Subsequently, the arrival of services to the Fog nodes (closest to the devices) will be modeled, and different strategies will be studied to distribute their processing between the two levels of the architecture. The behavior of different alternatives will be studied, analyzing the corresponding cost, as well as the total delay, taking into account different characteristics for the considered nodes: processing capacity, cost, etc.

Índice general

Índice de figuras

Índice de tablas

Índice de acrónimos

AWS Amazon Web Services.

ECDF *Empirical Cumulative Distribution Function*.

FPM *Fast Price Model*.

HTTP *Hypertext Transfer Protocol*.

IIoT *Industrial* IoT.

IoT *Internet of Things*.

IP *Internet Protocol*.

LXC *Linux Containers*.

MV Máquinas Virtuales.

SPM *Slow Price Model*.

TFM Trabajo Fin de Master.

1 Introducción

En primer lugar, en este capítulo se explica la motivación que impulsa a la realización de este Trabajo Fin de Master (TFM). Posteriormente, la siguiente sección expone los objetivos principales planteados al inicio del trabajo. Finalmente, el último apartado indica la estructura del documento, mencionando la información incluida en cada uno de los capítulos que lo conforman.

1.1. Motivación

En la actualidad, los diferentes servicios en la nube ofertados por los principales proveedores de servicios (Microsoft Azure, Amazon Web Services (AWS) y Google Cloud) son ampliamente utilizados, tanto por empresas de distintos sectores como por particulares e instituciones a nivel global. Si bien en los últimos años la demanda de estos servicios ha tenido una tendencia ascendente, con la progresiva instauración de servicios *Internet of Things* (IoT) [1,2] e *Industrial* IoT (IIoT) [3], con un crecimiento que parece impulsado por la implantación de las primeras redes de nueva generación 5G, se espera que esta demanda se mantenga, e incluso aumente, en los próximos años, ya que estos servicios pueden aprovechar muchas de las características de la computación en la nube.

Además, en la actualidad, los principales proveedores proporcionan servicios mejorados, y su catálogo es más diverso que hace unos años, por lo que empresas que todavía no han realizado una migración de sus modelos *on-premise* o locales hacia soluciones que utilicen parcial o totalmente servicios en la nube (debido a limitaciones impuestas por su modelo de negocio, requisitos tecnológicos o económicos, etc.) disponen de más opciones para realizar ese cambio, aprovechando así las principales ventajas ofrecidas por las soluciones *Cloud*.

Entre estas principales ventajas que presentan los servicios en la nube respecto a soluciones tradicionales o locales se encuentran la flexibilidad, la escalabilidad o el pago por uso. Sin embargo, esta tecnología también presenta algunas desventajas, como pueden ser el incumplimiento de los estrictos requerimientos de latencia de algunas aplicaciones, debido a las limitaciones de la red, la total dependencia de un tercero (el proveedor de servicios) o el coste variable de los servicios utilizados, principalmente, dependiendo de la demanda global.

Por ello, ha surgido un nuevo modelo denominado *Fog Computing* que, en un ámbito IoT, consiste en dotar o acercar parte de las capacidades proporcionadas por los servicios en la nube a los dispositivos. Así, combinando ambas soluciones, se pueden desplegar soluciones IoT-Fog-Cloud, que aprovechen las ventajas de ambos modelos y alternen entre ambos, dependiendo del tipo de servicio que se necesite

procesar, los requisitos del mismo o el coste asociado a cada una de las soluciones en un preciso instante temporal.

1.2. Objetivos

Con el objetivo de simular despliegues que sigan la arquitectura descrita y analizar sus características para comprobar su viabilidad, y establecer las configuraciones óptimas para los distintos dispositivos involucrados, en este TFM se propone el desarrollo de un demostrador de arquitectura tres capas IoT-Fog-Cloud. Así, los distintos objetivos a alcanzar durante el desarrollo del trabajo son los siguientes:

- Recopilación y presentación de información relevante presente en la literatura actual.
- Desarrollo del demostrador, utilizando una tecnología ligera, portable y disponible en un único dispositivo.
- Diseño y programación de diferentes aplicaciones software que cumplan las funciones características de los nodos *Fog* y *Cloud*.
- Inclusión de un emulador de enlace en la plataforma a desarrollar a partir del que realizar las comunicaciones entre nodos.
- Obtención de resultados relevantes utilizando la plataforma desarrollada, para comparar así diferentes configuraciones aplicables en las arquitecturas IoT-Fog-Cloud.
- Presentación de unas conclusiones finales, obtenidas a partir de los resultados, relacionándolas con lo expuesto en la literatura.
- Presentar las posibles líneas de desarrollo abiertas, que permitan mejorar la plataforma desarrollada.

1.3. Estructura

Este documento consta de cinco capítulos en los que se incluye la siguiente información:

- El primer capítulo expone la motivación de este trabajo, el objetivo principal con el que se desarrolla y la estructura que presenta el documento.
- El segundo capítulo explica los fundamentos teóricos necesarios para la comprensión del trabajo, explicando los conceptos principales asociados a las arquitecturas IoT-Fog-Cloud, así como diferentes aspectos utilizados en el proyecto.
- El tercer capítulo recoge la explicación de todas las herramientas y tecnologías utilizadas en el diseño del demostrador, además de exponer todo su desarrollo e implementación.

- El cuarto capítulo presenta los resultados obtenidos con el demostrador desarrollado, comparando diferentes configuraciones, en las que se van variando los principales parámetros que caracterizan tanto los dispositivos IoT como los nodos *Fog* y *Cloud*.
- El quinto y último capítulo expone las conclusiones obtenidas durante el desarrollo de todo el trabajo, así como diferentes líneas futuras que pueden surgir a partir del mismo.

2 | Conceptos Previos

La plataforma desarrollada durante este trabajo permite emular despliegues que siguen la arquitectura de tres capas IoT-Fog-Cloud. Para facilitar la comprensión del documento y contextualizar el trabajo, el segundo capítulo explica este tipo de arquitecturas. Posteriormente, se describe el término *containerization*, utilizado para el despliegue de la plataforma, el emulador de enlace empleado para las comunicaciones realizadas y, por último, el modelo de precios implementado.

2.1. Arquitectura IoT-Fog-Cloud

Como se ha mencionado en la introducción, los servicios IoT e IIoT son cada vez más comunes y su crecimiento parece verse impulsado con la instalación de las primeras redes de nueva generación 5G [4]. Gracias a las mejoras que presenta 5G respecto a las redes 4G en términos de latencia, disponibilidad, fiabilidad o densidad de conexión entre otros [5], muchos sectores y aplicaciones que se veían limitados por los requisitos en alguno de estos aspectos, abren la puerta al uso los servicios IoT.

En la actualidad, para poder manejar la gran cantidad de datos generada por las redes IoT, la solución más habitual se basa en la computación en la nube [6, 7] que, junto con técnicas de *Big Data*, permite procesar la información generada por, prácticamente, cualquier solución IoT. Sin embargo, con la tendencia creciente de estos servicios y su incorporación en nuevos sectores o aplicaciones con requisitos más exigentes, la computación en la nube puede no ser suficiente en términos de retardo, consumo energético o coste, entre otros.

El término *Fog computing* surge, en parte, como solución a las restricciones asociadas a la computación en la nube, ya que, su finalidad consiste en acercar parte de los servicios proporcionados por esta (principalmente capacidad de procesamiento y almacenamiento) a los dispositivos [8–10]. Así, muchas de las limitaciones que presenta la computación en la nube desaparecen. En primer lugar, el retardo introducido por las redes de comunicaciones a la hora de transmitir los datos generados por los dispositivos IoT a los nodos *Cloud* se elimina. Además, aunque la computación *Fog* introduce otro tipo de costes, se elimina la dependencia (o parte de ella, si se utiliza una solución combinada) de los proveedores de servicios en la nube, y por lo tanto, los costes asociados.

La plataforma desarrollada durante este trabajo se basa en una combinación de ambas soluciones, *Fog* y *Cloud computing*. Así, implementando una lógica de decisión (que puede estar basada en costes, retardos, almacenamiento, disponibilidad, etc.) se aprovechan las ventajas de ambas soluciones según convenga en cada momento y situación. Esto permite, por ejemplo, procesar la información más sensible

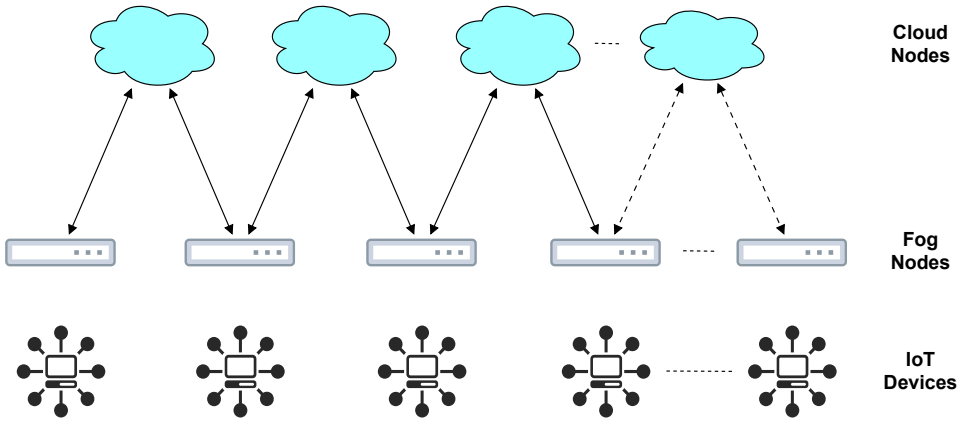


Figura 2.1: Diagrama general de las arquitecturas de tres capas IoT-Fog-Cloud.

en términos de retardo o seguridad en los nodos *Fog*. Así, la información no viajaría por la red, que podría ser el cuello de botella en términos de retardo, además de un punto vulnerable en cuanto a la seguridad. Por otro lado, las grandes cantidades de información que requieran una capacidad de cómputo muy elevada pueden ser una buena opción para procesar en los nodos *Cloud*, ya que la capacidad disponible en los nodos *Fog* será limitada.

Como un punto inicial, la política de decisión implementada en esta plataforma se basa en el precio, es decir, elegir el nodo en el que procesar la información en cada instante de tiempo para optimizar el coste, siempre teniendo en cuenta su disponibilidad y capacidad. La imagen de la Figura 2.1 muestra un esquema general de la arquitectura tres capas IoT-Fog-Cloud [11].

Aunque es cierto que ya existen algunas plataformas desarrolladas que presentan la arquitectura tres capas utilizada en este trabajo [12–14], el desarrollo e implementación de una plataforma a medida posibilita añadir funcionalidades específicas, según los requisitos propios asociados a este proyecto, como pueden ser la inclusión de un modelo de precios o un emulador de red determinado. Además, la plataforma desarrollada no es una solución cerrada, por lo que posibilita la incorporación de nuevas funcionalidades según surjan nuevas necesidades en el futuro.

2.2. *Containerization*

En el pasado, el desarrollo de cualquier aplicación o implementación software requería de una máquina física, y las diferencias existentes entre ellas, suponían un problema a la hora de desarrollar aplicaciones portables, adaptadas a los distintos sistemas operativos, o incluso, versiones de los mismos.

Con la aparición de las Máquinas Virtuales (MV), definidas como un software capaz de emular una máquina física y ejecutar cualquier programa o aplicación en ella, la portabilidad de aplicaciones entre diferentes máquinas dejó de ser un problema, ya que en cualquier máquina física pueden existir una o varias MV, emulando uno o varios sistemas operativos. Además de la portabilidad, las MV presentan otras ventajas, como la recuperación ante errores o un nivel de aislamiento de la máquina física que proporciona un nivel de seguridad adicional. Sin embargo, las MV comparten los recursos con las máquinas físicas en las que se ejecutan, lo que implica una sobrecarga adicional y, por lo tanto, una caída del rendimiento.

Tabla 2.1: Comparativa entre máquinas virtuales y contenedores.

	Máquinas virtuales	**Contenedores**
Aislamiento	*Hardware*	Sistema operativo
Sistema operativo	Separado	Compartido
Tiempo de inicio	Largo	Corto
Uso de recursos	Elevado	Bajo
Imágenes pre-construidas	Difíciles de encontrar	Disponibles
Imágenes personalizadas	Difíciles de generar	Fáciles de generar
Tamaño	Mayor porque contiene el SO completo	Menor porque comparte el SO con el *host*
Movilidad	Fácil de mover a un nuevo *host*	Se destruyen y recrean muy fácilmente
Tiempo de creación	Largo	En segundos

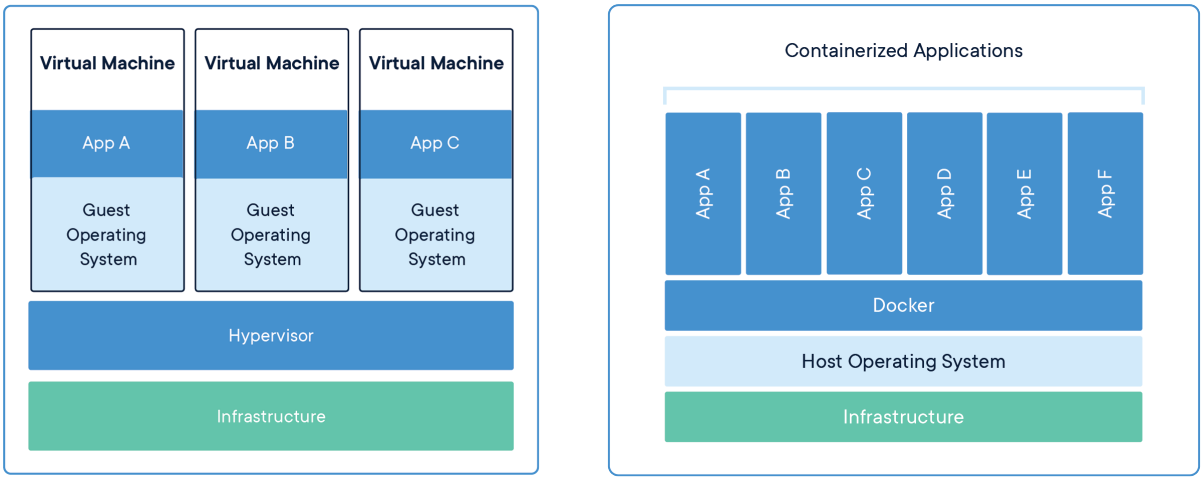
Aunque las MV sigan siendo ampliamente utilizadas, con el objetivo de conseguir una solución con lógica similar, pero evitando la desventaja de la sobrecarga producida en el *host*, aparecieron los contenedores, definidos como una lógica de abstracción en la capa de aplicación, capaces de ejecutar cualquier servicio, y cuya principal característica consiste en la compartición del kernel con el sistema anfitrión, reduciendo así la sobrecarga presente con las MV. Con el objetivo de aclarar las diferencias entre las MV y los contenedores, la Tabla 2.1 resume las características principales de ambas soluciones [15].

Por otro lado, si bien existen varias soluciones de *containerization* (Docker, *Linux Containers* (LXC), RKT, etc.) la más popular, y la utilizada en este trabajo, es Docker. Así, la Figura 2.2 muestra una comparativa del despliegue de aplicaciones en MV y en contenedores Docker, definidos como "una abstracción en la capa de la aplicación que empaqueta el código y las dependencias juntos. Se pueden ejecutar varios contenedores en la misma máquina y compartir el kernel del sistema operativo con otros contenedores, cada uno ejecutándose como procesos aislados en el espacio del usuario. Los contenedores ocupan menos espacio que las MV (las imágenes de los contenedor suelen tener un tamaño de decenas de MB) y pueden manejar más aplicaciones" [1].

Por ello, si se recopilan todas las características de la *containerization* expuestas en esta sección, su utilización en el desarrollo de la plataforma permite conseguir una solución ligera, portable, adaptable a cualquier máquina física (en este caso instalando Docker) y con gran facilidad y velocidad tanto en el despliegue como en la eliminación del entorno completo.

Así, con estas características, las posibilidades que permite la *containerization* son muy amplias. En el caso de la plataforma desarrollada, la estructura IoT-Fog-Cloud explicada previamente se ha implementado utilizando un contenedor por cada nodo necesario en el despliegue, bien sea de tipo *Fog* o *Cloud*. Para ello, dentro de cada contenedor desplegado se necesitan lanzar una serie de programas que implementan la lógica de estos nodos y, por lo tanto, actúen como tal (el *software* desarrollado para emular nodos *Fog* y *Cloud* se explica en la Sección 3.3). Además, cada uno de los tipos de nodos implementados se crean a partir de imágenes Docker personalizadas (ver Sección 3.2), que son archivos compuestos por varias instrucciones (cada una crea una capa diferente), que se utilizan para ejecutar código en un contenedor. El hecho de trabajar por capas permite modificar y crear imágenes distintas, así como versiones de las mismas

[1] https://www.docker.com/resources/what-container



(a) (b)

Figura 2.2: Diferencias entre el despliegue de aplicaciones en MV (a) y contenedores Docker (b).

en un periodo de tiempo muy reducido, ya que si alguna de las capas es compartida por otra imagen ya descargada, esta se reutiliza. Además, la plataforma Docker permite subir las imágenes personalizadas a un repositorio accesible desde cualquier máquina con conexión a internet (Docker Hub [2]), por lo que la portabilidad de la plataforma es inmediata.

Por otro lado, en cuanto al despliegue de la plataforma, existen varias opciones. En un entorno real, los diferentes nodos de un despliegue no se encuentran en una misma máquina física. Este escenario se puede replicar en la plataforma de pruebas desarrollada, ya que la tecnología Docker permite desplegar todos sus contenedores de forma distribuida. Así, aprovechando esta característica, se pueden realizar despliegues de la plataforma de pruebas en entornos reales, consiguiendo que las comunicaciones entre los diferentes nodos compartan la misma red que verían los nodos de un despliegue real.

Además, la *containerization* también permite el despliegue de todos los nodos o contenedores en una misma máquina física, opción utilizada durante el desarrollo de este trabajo. Sin embargo, utilizando esta solución, las comunicaciones entre los diferentes nodos no sufren ninguna alteración similar a las encontradas en un entorno real, ya que que no salen de la máquina local. Para poder obtener resultados que capturen un comportamiento realista, se ha incorporado un emulador de enlace en la plataforma, que permite añadir diferentes características a las conexiones existentes entre los contenedores.

2.3. Emulador de enlace

Como se ha mencionado, una de las principales características de la plataforma desarrollada consiste en la posibilidad de realizar un rápido despliegue en una sola máquina física. Sin embargo, esto implica que todos los contenedores (nodos *Fog* y *Cloud*) se encuentran en la misma red, siendo accesibles entre ellos sin sufrir ningún retardo u otras características propias de una red de comunicaciones real. Aunque la tecnología Docker permite aislar y crear distintas redes donde alojar los contenedores, configurando la accesibilidad de forma similar a la encontrada en una configuración *Internet Protocol* (IP), el hecho de

[2] https://hub.docker.com/

que las comunicaciones no salgan de la misma máquina física sigue limitando la simulación de una red real.

Para solucionar este problema, existen varios emuladores de red o enlace que pueden encajar en la plataforma, siendo los más conocidos ns-3 [3], GNS3 [4] y Mininet [5], los cuales se han utilizado en múltiples investigaciones e implementaciones con una temática similar a la propuesta en este TFM. Algunos ejemplos que se pueden encontrar en la literatura son [16–18].

Sin embargo, con el objetivo de utilizar una solución sencilla y ligera, el emulador de enlace empleado en este trabajo ha sido Mahimahi [19], definido como "un conjunto de herramientas ligeras para desarrolladores, autores de sitios web y diseñadores de protocolos de red que proporciona mediciones precisas al grabar y reproducir contenido *Hypertext Transfer Protocol* (HTTP) en condiciones de red emuladas. Cada herramienta Mahimahi genera un contenedor ligero, generalmente conectado al exterior a través de un dispositivo de red virtual que observa los paquetes en tránsito y emula un comportamiento deseado" [6].

Así, en el caso de la plataforma desarrollada, Mahimahi permite emular redes reales de diferentes proveedores de telecomunicación entre los nodos de la plataforma, ya que la herramienta admite ficheros que contengan las trazas necesarias para emular esas redes. Además, todas las herramientas proporcionadas por Mahimahi pueden implementarse de manera anidada, lo que implica que, además de incluir las trazas de los operadores, un usuario de la plataforma puede añadir características específicas de retardo o pérdida de paquetes a sus comunicaciones.

2.3.1. Modelo de precios

Aunque los proveedores de servicios en la nube, como norma general, ofrezcan capacidad de cómputo a sus usuarios siguiendo una política de pago por uso, las variaciones en la demanda de estos servicios dependiendo del día y la hora en la que se soliciten, suponen cambios constantes en el precio final de los mismos. Además, los principales proveedores de servicios han lanzado un nuevo producto, denominado *spot instances* por AWS [7], *preemptible VM instances* por Google Cloud [8], o *Azure spot VM* por Microsoft Azure [9], cuya definición puede encajar en los requisitos de las plataformas IoT-Fog-Cloud. Este nuevo producto consiste en ofrecer instancias iguales a las disponibles tradicionalmente cuando la demanda de las últimas no es muy exigente, pero a un precio muy reducido. Sin embargo, si la demanda de recursos por los servicios tradicionales aumenta, las instancias a precio reducido pueden detenerse. Es decir, los proveedores de servicio ofrecen la posibilidad de utilizar el exceso de su capacidad en determinados momentos de baja demanda a precios reducidos, pero no garantizan su continua disponibilidad. Así, estas nuevas instancias pueden encajar en las plataformas IoT-Fog-Cloud, diferenciando entre los distintos tipos de servicios a procesar, y enviando a los nodos *Cloud* los que presenten características compatibles con las mismas.

[3] https://www.nsnam.org/
[4] https://www.gns3.com/
[5] https://github.com/mininet/mininet/wiki/Documentation
[6] http://mahimahi.mit.edu/
[7] *https://aws.amazon.com/ec2/spot*
[8] *https://cloud.google.com/compute/docs/instances/preemptible*
[9] *https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/use-spot*

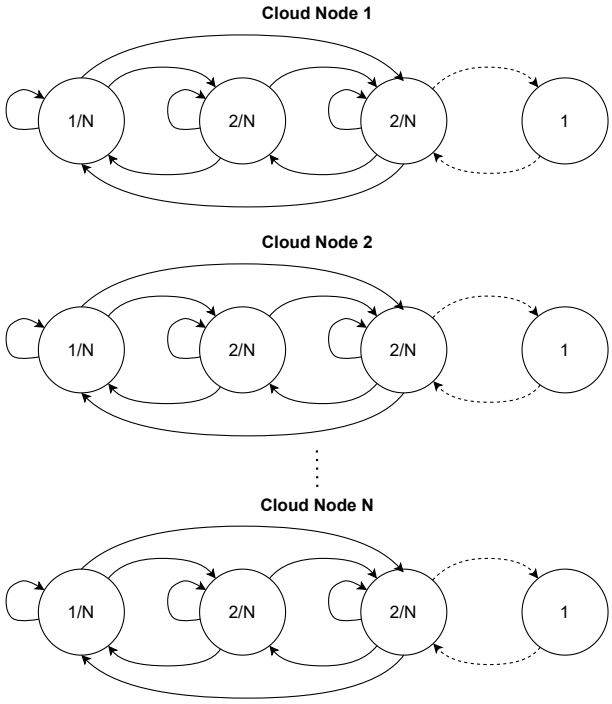


Figura 2.3: Cadena de Markov del modelo de precios implementado para el coste de procesado en los nodos *Cloud*.

Por todo ello, en la plataforma desarrollada en este TFM se ha implementado el modelo de precios expuesto en [20], permitiendo así dotar a los nodos *Cloud* con un precio por uso variable en el tiempo según un parámetro de configuración dado, consiguiendo de este modo que la política de decisión basada en el coste proporcione unos resultados más interesantes.

Así, cada uno de los nodos *Cloud* desplegados en la plataforma tiene asociada una cadena de Markov independiente, que cambia de estado periódicamente según se haya configurado previamente (desde segundos hasta días, dependiendo de las necesidades de la simulación). Cada uno de los estados de la cadena representa el coste actual del procesado en su nodo *Cloud* asociado, que se establece con dependencia del número total de estados (N), desde 1/N hasta 1 (ver Figura 2.3).

Además, se han implementado las dos opciones propuestas en el artículo: *Slow Price Model* (SPM) y *Fast Price Model* (FPM). El primero limita las transiciones entre los estados de tal forma que desde el estado actual solo se puede saltar a sus contiguos y se puede definir como:

$$P_{ij} = \begin{cases} 0 & \delta > 1 \\ \dfrac{1}{2\,\mathcal{H}(2) - 1} \cdot \dfrac{1}{\delta + 1} & i = 1 \ldots N - 2,\ \delta \leq 1 \\ \dfrac{1}{\mathcal{H}(2)} \cdot \dfrac{1}{\delta + 1} & i = 0,\ N - 1,\ \delta \leq 1 \end{cases} \tag{2.1}$$

donde $\mathcal{H}(n)$ es el n-ésimo número harmónico.

Por otro lado, el segundo no presenta ninguna restricción en las transiciones entre estados, por lo que

desde el estado actual se puede saltar a cualquier otro dentro de la cadena.

$$P_{ij} = \frac{1}{\mathcal{H}(i+1) + \mathcal{H}(N-i) - 1} \cdot \frac{1}{\delta + 1} \tag{2.2}$$

3 | Desarrollo e implementación

Una vez explicados los principales términos y herramientas utilizadas en el desarrollo de este trabajo, exponiendo además literatura de referencia sobre los mismos, este capítulo explica cómo se han utilizado estas herramientas para desarrollar la plataforma de pruebas implementada en este proyecto.

En primer lugar, en las Secciones 3.1 y 3.2 se describirán las herramientas adoptadas: Mahimahi y Docker. Posteriormente, en la Sección 3.3 se describirá la implementación llevada a cabo detallando el uso que se hace de estas herramientas.

3.1. Mahimahi

Así, aunque las razones por las cuales se ha elegido Mahimahi como el emulador de enlace empleado en esta plataforma se han enumerado en el Capítulo 2, a continuación se explica cómo se ha integrado esta herramienta en la plataforma.

En primer lugar, sabiendo que las herramientas disponibles con Mahimahi crean contenedores, los cuales modifican todo el tráfico que los atraviesa, es necesario que todo el tráfico cursado entre los diferentes nodos *Fog* y *Cloud* de la plataforma crucen uno de estos contenedores. Así, cualquier comunicación realizada en la plataforma desarrollada obtendrá características similares a las sufridas en una red de comunicaciones real. Para conseguir este objetivo se han planteado varias soluciones válidas, cada una con una serie de ventajas y desventajas:

- Un único contenedor Mahimahi por el cual redirigir todo el tráfico cursado - Esta primera opción presenta una clara ventaja frente al resto, y es que solo es necesario desplegar un contenedor Mahimahi en toda la plataforma. Sin embargo, esto presenta varias limitaciones. Primero, es necesario configurar reglas de red por cada uno de los nodos para redirigir todo el tráfico. Segundo, todos los nodos de la plataforma verían la misma red, comportamiento que se aleja de la realidad ya que, en un despliegue real, estos no se encuentran en el mismo punto.

- Un contenedor Mahimahi anidado en cada contenedor *Cloud* - Esta opción pierde la ventaja de la anterior, ya que es necesario desplegar tantos contenedores Mahimahi como nodos *Cloud* existan en la plataforma. Sin embargo, por cada uno de estos contenedores se pueden configurar redes distintas, por lo que cada una de las comunicaciones con los nodos *Cloud* presenta características de red diferentes.

- Un contenedor Mahimahi anidado en cada contenedor *Fog* - Esta ha sido la opción implementada en

esta plataforma. Es similar a la anterior, ya que se pueden configurar tantas redes como nodos *Fog* se desplieguen en la plataforma, y además no es necesario crear ninguna regla de red adicional. Como ventaja adicional, permite definir condiciones de red específicas para cada nodo *Fog*, lo que data a la emulación de más realismo al permitir modelar canales inalámbricos diferentes en los nodos *Fog*, aspecto importante en escenarios IIoT.

Así, por cada contenedor Docker que simule un nodo *Fog*, un contenedor Mahimahi se despliega en su interior, y dentro de este, es donde se ejecutan los programas con la lógica asociada a estos nodos. De este modo, todo el tráfico existente en los nodos *Fog* que tenga que ser enviado a cualquier nodo *Cloud*, siempre contará con las características de red impuestas para ese nodo, ya que para salir del contenedor Mahimahi, atravesará la interfaz virtual asociada al mismo.

Con la explicación hasta este punto, la inclusión de Mahimahi en la plataforma desarrollada parece sencilla y directa. Sin embargo, se ha podido comprobar que la herramienta posee ciertas limitaciones que se han tenido en cuenta a la hora de utilizarla. En primer lugar, para ejecutar la herramienta, es necesario utilizar el sistema operativo Ubuntu 14.04 o una versión superior del mismo. Considerando que todas las imágenes desarrolladas para los diferentes nodos de la plataforma se han basado en Ubuntu 18.04 (ver Sección 3.2), esta limitación no implica ningún problema adicional. En segundo lugar, ninguna de las herramientas disponibles con Mahimahi permite su ejecución con el usuario *root* (los desarrolladores de la plataforma han incluido esta limitación imposibilitando el despliegue de cualquier contenedor Mahimahi utilizando el usuario *root*), por lo que ha sido necesaria la creación de un usuario adicional en todos los contenedores en los que se usa Mahimahi. Por último, la herramienta utiliza algunos directorios protegidos de la máquina en la que se implementa para emular las redes de comunicaciones. Como el trabajo se ha desarrollado utilizando la plataforma Docker, por defecto todos los contenedores se construyen buscando la característica de ligereza, lo que implica que muchos de los directorios que necesita Mahimahi no son incluidos por defecto. Para evitar este problema y considerando que se ha desarrollado una plataforma de pruebas, donde la seguridad no es un aspecto crítico, todos los contenedores se han ejecutado en modo privilegiado, permitiendo así el uso de todos los directorios de la máquina *host*, incluyendo los que necesita la herramienta Mahimahi para funcionar correctamente.

3.2. Contenedores Docker

Como se ha mencionado en el capítulo anterior, para realizar el despliegue de los diferentes nodos de la plataforma se han utilizado contenedores Docker, los cuales actúan como nodos de distintos tipos. Si bien dentro de un mismo contenedor se podría realizar el despliegue de varios nodos (incluso de la plataforma completa si la implementación elegida fuese diferente), en este caso cada uno de los contenedores Docker actúa como un único nodo, bien sea *Fog*, *Cloud*, o el nodo *Master* (explicados en la Sección 3.3). Esta implementación presenta varias ventajas respecto a otras opciones, como pueden ser la facilidad de implementar varios tipos de despliegues (creando y lanzando plataformas de características diferentes en cuestión de segundos), la posibilidad de automatizar la creación, despliegue y destrucción de todos los nodos de la plataforma (utilizando Docker *Compose*), o la posibilidad de aprovechar la red creada por Docker automáticamente para realizar todas las comunicaciones necesarias entre nodos (incluyendo

Mahimahi para emular características de red reales).

Así, los tres tipos de nodos distintos existentes en la plataforma se crean a partir de imágenes personalizadas basadas en Ubuntu 18.04 (ver Anexos A, B y C). Para ello, en los tres casos, el primer paso consiste en importar la imagen de Ubuntu 18.04 para poder personalizarla según los requerimientos de cada nodo.

Por defecto, siguiendo la política de ligereza asociada a la tecnología Docker, la imagen importada cuenta con los paquetes mínimos para poder funcionar. Sin embargo, para desplegar tanto los programas desarrollados en Python, como implementar el emulador Mahimahi dentro de los contenedores, es necesario instalar varios paquetes adicionales, siendo esta la siguiente instrucción en cualquiera de las tres imágenes desarrolladas.

Una vez descargados todos los paquetes necesarios en cada una de las imágenes, los siguientes pasos no coinciden en los tres casos. Para la imagen de los nodos *Fog*, la siguiente instrucción es la encargada de descargar e instalar el emulador de enlace Mahimahi, ya que son estos nodos los únicos que lo necesitan. Además, debido a las restricciones impuestas por la herramienta (ver Sección 3.1), es necesario crear un usuario adicional (por defecto se utiliza el usuario *root*). Una vez creado el usuario con las características adecuadas, las últimas instrucciones son las asociadas a los programas Python desarrollados para emular un nodo *Fog*. Primero, se copian los programas desarrollados al contenedor Docker para, posteriormente, con las dos últimas instrucciones, estos se ejecuten una vez el contenedor ha sido desplegado. Es importante apuntar que la instrucción de copia de los programas Python debe contener en su primer argumento el *path* donde estos estén localizados. Además, cabe mencionar que la copia de archivos podría ser fácilmente remplazada por su descarga desde repositorios de código.

En cuanto a las imágenes de los nodos *Cloud* y el nodo *Master*, ambas se crean de forma similar. Como no es necesario instalar Mahimahi en los contenedores asociados a estas, después de instalar todos los paquetes necesarios, solo queda copiar los programas Python desarrollados correspondientes a cada una e indicar su ejecución en el momento de despliegue del contenedor.

A continuación se comentará la automatización del despliegue de todos los nodos y la ejecución de la plataforma de pruebas.

3.2.1. Docker *Compose*

Como se ha mencionado previamente, una de las características principales que presenta la *containerization* es la posibilidad de realizar despliegues en cuestión de segundos, así como terminarlos con la misma velocidad. Sin embargo, en el caso de la plataforma desarrollada, esta tiene muchos parámetros configurables, lo que presenta la clara ventaja de poder comparar muchos escenarios con características diferentes, pero implica la configuración de los mismos en cada una de las simulaciones.

Con el objetivo de automatizar y facilitar los despliegues, se ha creado un fichero en el cual indicar todos estos parámetros y con el cual realizar los despliegues con una sola instrucción. La herramienta que permite esta funcionalidad es Docker *Compose*, que se define como "una herramienta para definir y ejecutar aplicaciones Docker de varios contenedores. Con Docker *Compose*, se utiliza un archivo YAML para configurar los servicios de la aplicación. Después, con un solo comando, se crean e inician todos

Tabla 3.1: Lista de los parámetros configurables.

Parámetro	Unidades
Tiempo de simulación	segundos
path a Volúmenes	-
Trazas Mahimahi	-
# de nodos Fog & Cloud	Nodos
Capacidad de procesado de nodos Fog & Cloud	Bps
Longitud de paquetes	Bytes
Tasa de tráfico	Pkts/s
Tasa de *servicio*	Serv/s
Longitud de cola Fog	Bytes
# de estados de la cadena de Markov	Estados
Aproximación tradicional	si / no
Tipo de modelo de precios	FPM / SPM
Coste del nodo Fog	0 - 1

los servicios requeridos" [1]. Como ejemplo, en el Anexo D se expone un fichero YAML válido utilizado durante el trabajo.

A modo de resumen, la Tabla 3.1 recoge todos los parámetros y opciones configurables disponibles en la plataforma desarrollada.

Como se puede ver, la configuración incluye, además de las rutas de ficheros, parámetros que afectan al comportamiento de los nodos de forma diversa. Por un lado, se configura la capacidad de procesado de los nodos (*Fog* y *Cloud*), así como la memoria disponible en los nodos *Fog*. Este último parámetro no se incluye para los nodos *Cloud* ya que se considera que poseen memoria infinita. Además, se configura de forma detallada la generación de tráfico, la cual es una emulación del tráfico que provendría de dispositivos IoT. Por último, la configuración también incluye los parámetros necesarios para definir el comportamiento del modelo de precios descrito anteriormente. Todos estos parámetros son visibles dentro de los contenedores Docker como variables de entorno.

3.3. Programación de los nodos

Para implementar los diferentes tipos de nodos que se pueden desplegar en la plataforma, dentro de cada uno de los contenedores que actúen como tal, es necesario cargar y ejecutar uno o varios programas. Así, en esta sección se explican todos los programas desarrollados (utilizando Python) para dotar a los contenedores con las características propias de los distintos tipos de nodos, además de su funcionalidad.

[1] https://docs.docker.com/compose

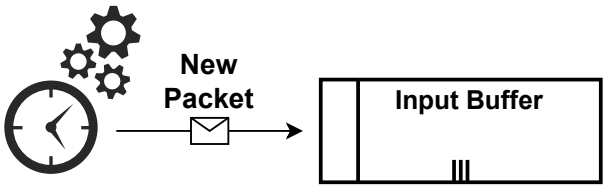


Figura 3.1: Generación de paquetes en los nodos *Fog*.

3.3.1. Nodos *Fog*

Como se ha mencionado a lo largo del documento, todos los nodos *Fog* tienen la posibilidad de lanzar un contenedor Mahimahi en su interior si una red con unas características específicas necesita emularse. Para ello, los contenedores que emulen nodos *Fog* necesitan dos programas, el primero, que ejecuta la herramienta Mahimahi, y un segundo que implementa toda la lógica asociada a los nodos *Fog* (procesado de una cola, envío de información, etc.).

Respecto al primero, es el responsable de ejecutar los comandos que se encargan de cambiar de usuario y ejecutar Mahimahi con los parámetros especificados en el fichero de configuración. Además, este programa se encarga de exponer los parámetros de configuración, presentes en el contenedor Docker, dentro del contenedor generado por Mahimahi. Estos parámetros incluyen la configuración del enlace emulado por Mahimahi, ya sea mediante retardos, trazas de operadores, o combinaciones de ambos.

Para ello, el programa comienza leyendo todas las variables involucradas. Después, ejecuta el comando encargado de cambiar de usuario y abre una *pipe* para ejecutar todos los comandos restantes dentro del usuario al que se acaba de acceder. A continuación, se preparan todos los comandos a ejecutar, incluyendo el encargado de establecer las variables globales, el que lanza Mahimahi con las especificaciones requeridas, y el que ejecuta el segundo programa dentro del contenedor Mahimahi, y por lo tanto, dentro del nodo *Fog*. Por último, se envían todos los comandos a través del *pipe* para que se ejecuten en el usuario previamente seleccionado. En este punto, el segundo programa se inicia, lo que implica que el nodo *Fog* se ha desplegado correctamente.

El segundo programa implementa toda la lógica asociada a un nodo *Fog*. Primero, en lugar de recibir el tráfico de los dispositivos IoT, considerando que se ha desarrollado una plataforma de pruebas, el programa emula la recepción del tráfico con una función (ver Figura 3.1). En futuras modificaciones, esta implementación se podría cambiar e incluir algunos dispositivos IoT generando y enviando tráfico real a los nodos *Fog*. Además, el tráfico generado emula la llegada de grupos de paquetes, a los que denominaremos *servicio*, que deben ser procesados conjuntamente, ya sea en un nodo *Fog* o *Cloud*. De esta manera, los paquetes pertenecientes a un *servicio* se almacenan en el nodo *Fog* hasta la recepción de todos, momento en el cual se inicia su procesado conjunto.

Así, la función mencionada genera tráfico (paquetes) siguiendo una distribución de Poisson, es decir,

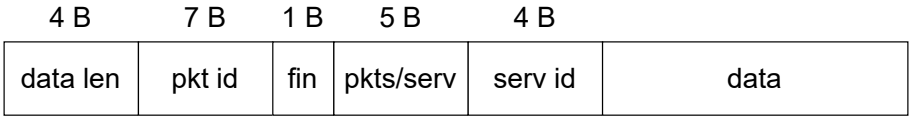


Figura 3.2: Formato de los paquetes.

con un tiempo entre nuevos paquetes exponencial negativo. Después, antes de almacenar los paquetes generados en un *buffer*, se incluyen algunas cabeceras (ver Figura 3.2) necesarias para poder realizar el seguimiento y finalmente el procesado de estos, bien sea en local o remoto. Estas cabeceras son:

- **data len**: longitud total en bytes del los datos del paquete.
- **pkt id**: identificador único del paquete. Comienza en 1 y se incrementa con la generación de cada nuevo paquete.
- **fin**: *flag* que sirve para identificar el último paquete enviado a los nodos *Cloud*. Siempre vale 0 y se cambia a 1 para indicar el fin de la simulación.
- **pkts/serv**: número de paquetes totales pertenecientes al *servicio*. Solo se indica en el primer paquete de los *servicios* enviados.
- **serv id**: identificador único del *servicio*. Comienza en 1 y se incrementa con la generación de cada nuevo *servicio*.

Finalmente, la función comprueba periódicamente si el tiempo total de simulación se ha completado, y si es así, genera un último paquete con un *flag* de fin activo, el cual es enviado a todos los nodos *Cloud* para indicar que ningún paquete adicional va a ser enviado.

La siguiente función se encarga de generar los *servicios*, los cuales siguen también una distribución de Poisson, al igual que la generación de tráfico. Para emular que un cierto número de paquetes pertenecen a un determinado *servicio*, se establece una tasa de generación de *servicios* (siempre menor que la tasa de generación de tráfico, ya que de lo contrario podrían existir *servicios* sin ningún paquete). Esta característica permite a la plataforma emular un procesado a nivel de *servicio*, consiguiendo un comportamiento más realista. Así, cuando llega un nuevo *servicio*, todos los paquetes que se encuentren actualmente en el *buffer* pasan a pertenecer a ese *servicio* y, dependiendo de la información proporcionada por el nodo *Master* (explicado en la Sección 3.3.3), todos esos paquetes son transferidos desde el *buffer* hacia la cola de procesado en el propio nodo *Fog*, o enviados al correspondiente nodo *Cloud* para que sean procesados de forma remota (ver Figura 3.3).

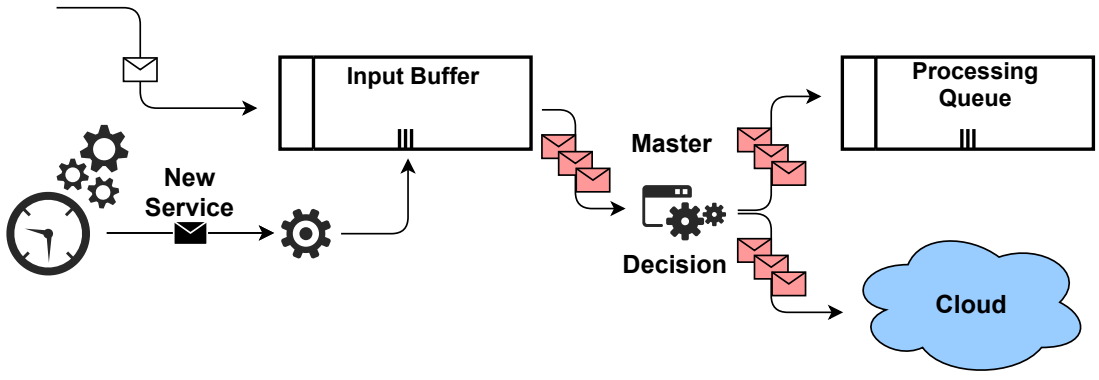


Figura 3.3: Llegada de nuevos *servicios* a los nodos *Fog*.

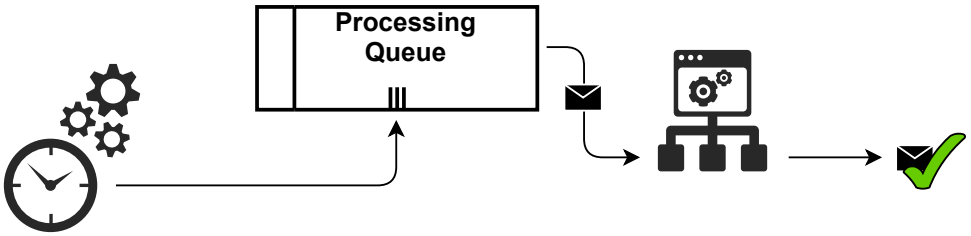


Figura 3.4: Procesado en los nodos *Fog* y *Cloud*.

La última función del programa es la responsable de procesar el tráfico localmente, en el nodo *Fog*. Para ello, esta espera hasta que un *servicio* completo entra en la cola de procesado y, entonces, comienza a procesar todos los paquetes de ese *servicio* de acuerdo a una tasa de procesado previamente configurada. Como la longitud de los paquetes también sigue una distribución de Poisson, el tiempo de procesado de estos también presenta una distribución exponencial negativa (ver Figura 3.4).

Para emular todos los tiempos de procesado y espera en cualquiera de las tres funciones, el código simplemente llama a la función *sleep* con el tiempo requerido en cada caso, pero no se realiza ninguna operación de procesado real con los datos, que son aleatorios. Además, las tres funciones se lanzan mediante diferentes hilos. Así, todas pueden trabajar y ejecutar su lógica en paralelo, de forma independiente, sin interferir la una con la otra. Esto conlleva la correcta sincronización entre hilos para permitir la transferencia segura de tráfico entre los *buffers* de llegada y procesado o entre el *buffer* de llegada y el envío a la red.

3.3.2. Nodos *Cloud*

Como se ha explicado previamente en esta sección, algunos de los *servicios* generados son procesados en los nodos *Fog*, mientras que otros son enviados a los nodos *Cloud*. Por ello, la plataforma cuenta con varios contenedores que implementan la lógica de un nodo *Cloud*.

En primer lugar, todos los nodos *Cloud* cuentan con una función encargada de recibir el tráfico enviado desde el otro extremo de la red y almacenarlo en un *buffer*. El esquema de la Figura 3.5 muestra cómo los paquetes son enviados desde los nodos *Fog* hasta los nodos *Cloud* a través de la red emulada con Mahimahi. Al igual que con las funciones explicadas previamente, esta también se implementa con un hilo independiente, el cual está continuamente en funcionamiento durante todo el periodo de simulación, comprobando si llega algún nuevo paquete.

Posteriormente, antes de procesar todos los paquetes recibidos, es necesario que en el nodo *Cloud* se espere hasta recibir un *servicio* completo. Para implementar esta funcionalidad, otra función simulada

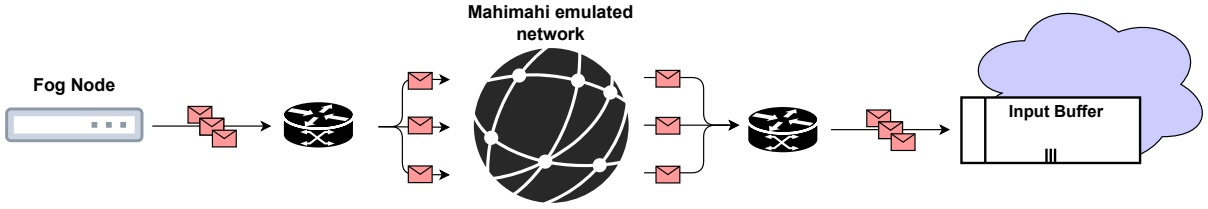


Figura 3.5: Camino desde los nodos *Fog* hasta los nodos *Cloud*.

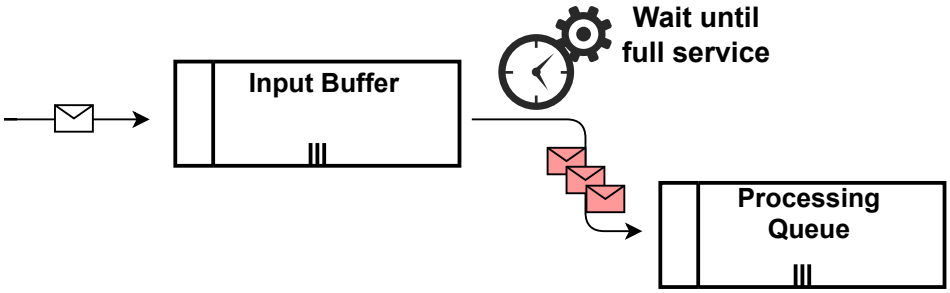


Figura 3.6: Esperar un *servicio* completo antes de mover los paquetes a la cola de procesado.

con otro hilo independiente se encarga de comprobar continuamente las cabeceras de todos los paquetes recibidos. Así, una vez que llega el *servicio* completo, esta mueve todos los paquetes pertenecientes a ese *servicio* desde el *buffer* hasta la cola de procesado del propio nodo *Cloud* (ver Figura 3.6).

Finalmente, siguiendo la misma lógica implementada en los nodos *Fog* (ver Figura 3.4), otro hilo independiente simula un procesador que se encarga de procesar los *servicios* que vayan entrando a la cola.

3.3.3. Nodo *Master*

A parte de los nodos *Fog* y *Cloud*, cualquier despliegue de la plataforma desarrollada cuenta con un nodo adicional llamado *Master*. Primero, con el fin de emular escenarios reales donde las decisiones se toman en base al coste de procesar los datos, bien sea local o remotamente, el modelo de precios explicado en la Sección 2.3.1 se ha implementado para funcionar en el nodo *Master* de cualquier despliegue. Además, este nodo emula una conexión con un proveedor de *servicios* en la nube que actualiza el coste de sus *servicios* (capacidad de procesado en los nodos *Cloud*) en tiempo real. Esto permite a la plataforma tomar decisiones sobre dónde procesar la información, en local o en remoto, en base al coste de todos los nodos disponibles en un determinado momento. Así, cada vez que a un nodo *Fog* le llega un nuevo *servicio* para procesar, este se lo notifica al nodo *Master*, el cual le responde con la información sobre dónde procesarlo.

Para emular este comportamiento, el nodo *Master* implementa dos procesos independientes. Primero, el mencionado modelo de precios, que se ejecuta tantas veces como nodos *Cloud* se hayan desplegado. Así, todos ellos cuentan con una cadena de Markov independiente que los dota de un coste asociado por

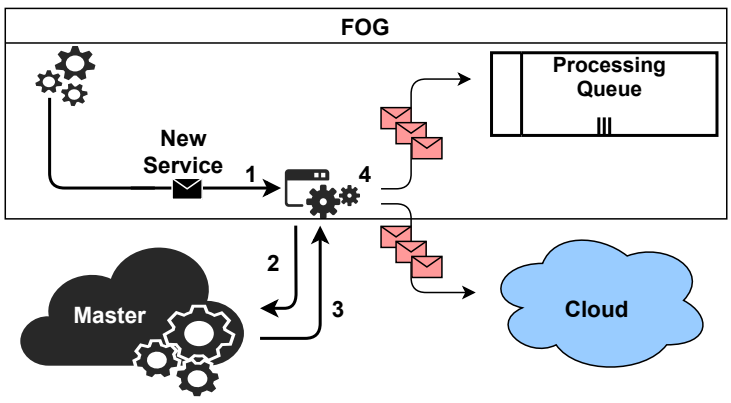


Figura 3.7: Secuencia de los nuevos *servicios* en los nodos *Fog*.

unidad de tiempo diferente.

Segundo, el nodo *Master* tiene un enlace abierto con cada uno de los nodos *Fog*. Como se ha mencionado, esto permite a la plataforma decidir dónde procesar los *servicios* cada vez que llegan, considerando el coste de todos los nodos *Cloud*, además del estado de los nodos *Fog* (estos tienen una cola de procesado finita, y puede estar ocupada). Así, en el momento que un *servicio* completo llega al nodo *Fog*, este se lo notifica al *Master*, el cual decide dónde procesarlo. Después, el nodo *Master* le comunica la decisión de vuelta al nodo *Fog*, donde el *servicio* pasa a la cola de procesado local, o es enviado al nodo *Cloud* correspondiente (ver Figura 3.7).

4 | Resultados

Tras describir la implementación realizada, en este capítulo se muestra el análisis obtenido mediante la plataforma en varias configuraciones. Este análisis tiene dos objetivos: en primer lugar validar el correcto funcionamiento de la implementación, y adicionalmente se buscan analizar las ventajas de esquemas dinámicos de selección de procesado basados en precio. La Tabla 4.1 muestra la configuración utilizada para todas las simulaciones recogidas, indicando los parámetros que se han variado.

Así, con el objetivo de comparar una aproximación tradicional de las soluciones *Cloud* (toda la información es enviada a un nodo *Cloud* en particular, el cual procesa todos los datos) con una solución *Fog-Cloud*, se han simulado diferentes escenarios variando las capacidades del nodo *Fog*, así como el tráfico. En concreto, se han considerado las siguientes configuraciones del nodo *Fog*:

- Tradicional - Todos los datos son procesados en un único nodo *Cloud* sin aplicar política de precios.
- No *Fog* - Todos los datos son procesados en diferentes nodos *Cloud*, considerando el precio de los mismos.
- Cola *Fog*: 0 Bytes - Se puede procesar información en el nodo *Fog* pero no se implementa cola de espera local.
- Cola *Fog*: 250000 Bytes - Se puede procesar información en el nodo *Fog* y se dispone de una cola de 250000 Bytes.
- Cola *Fog*: 500000 Bytes - Se puede procesar información en el nodo *Fog* y se dispone de una cola de 500000 Bytes.
- Cola *Fog*: 750000 Bytes - Se puede procesar información en el nodo *Fog* y se dispone de una cola de 750000 Bytes.
- Cola *Fog*: 1000000 Bytes - Se puede procesar información en el nodo *Fog* y se dispone de una cola de 1000000 Bytes.

Además, para cada configuración de memoria se han simulado distintas capacidades de procesado en los nodos *Fog*: 1000, 2000, 5000 y 10000 Bytes/segundo o, comparándolas con los nodos *Cloud* 1 %, 2 %, 5 % y 10 % de la capacidad de procesado de los nodos *Cloud*, que se ha fijado en 1e5 Bps, tal como se indica en la Tabla 4.1. Finalmente, el conjunto completo de simulaciones se ha replicado dos veces, modificando la tasa de generación de tráfico en 10 y 100 paquetes/segundo. Finalmente, se ha realizado un

Tabla 4.1: Configuración del primer conjunto de simulaciones.

Parámetro	**Valor**
Tiempo de simulación	12000 s
Trazas Mahimahi	TMobile-LTE-short
# Nodos *Cloud*	4 Nodos
# Nodos *Fog*	1 Nodo
Capacidad de procesado en nodos *Cloud*	100000 Bps
Capacidad de procesado en nodos *Fog*	{1000, 2000, 5000, 10000} Bps
Longitud de paquetes media	200 Bytes
Tasa de tráfico	{10, 100} Pkts/s
Tasa de *servicio*	0.083 Serv/s
Longitud de la cola en nodos *Fog*	{0, 250000, 500000, 750000, 1000000} Bytes
# Estados en cadena de Markov	50 Estados
Aproximación tradicional	si & no
Tipo de modelo de precios	FPM
Coste en nodos *Fog*	0

análisis con valores de procesado y generación de tráfico más elevados a fin de obtener un comportamiento más cercano a la realidad. En todos los casos, se han realizado emulaciones de 200 minutos para cada una de las configuraciones.

Uno de los principales objetivos de la plataforma desarrollada es poder comparar el coste de procesar la información generada utilizando diferentes configuraciones, tanto en escenarios tradicionales como en aquellos que siguen la arquitectura *Fog-Cloud*.

Así, la Figura 4.1 muestra la *Empirical Cumulative Distribution Function* (ECDF) del coste por *servicio* para una tasa de generación de 10 pkts/s y todas las tasas de procesado mencionadas, teniendo en cuenta que el coste asociado a cada nodo *Cloud* sigue el modelo de precios explicado en la Sección 2.3.1 y considerando un coste nulo en los nodos *Fog*. En las cuatro gráficas expuestas se compara el coste medio de procesar todos los *servicios* generados durante un periodo de tiempo dado, considerando todos los escenarios mencionados previamente.

En primer lugar, se aprecia que la configuración 'tradicional' (todos los paquetes son procesados en el mismo nodo *Cloud* sin considerar el coste asociado a ningún otro nodo) es la que tiene el mayor coste medio por *servicio*. Como en los cuatro escenarios expuestos solo se ha variado la capacidad de procesado local, el resultado de la solución 'tradicional' no presenta cambios en ningún caso. Este comportamiento es el esperado ya que, en este despliegue, no se ha implementado ninguna política de costes. Sin embargo, en cuanto se tiene en cuenta el coste de todos los nodos *Cloud* disponibles, incluso sin incorporar capacidad de procesado en el nodo *Fog* y, por lo tanto, procesando todos los paquetes de forma remota (solución 'No *Fog*'), el coste medio por *servicio* se reduce considerablemente, aunque tampoco varía de una configuración a otra, ya que todos los *servicios* se procesan en los nodos *Cloud*, donde ningún parámetro cambia en las diferentes configuraciones.

En cuanto a las simulaciones que implementan capacidad de procesado y una cola de espera en el

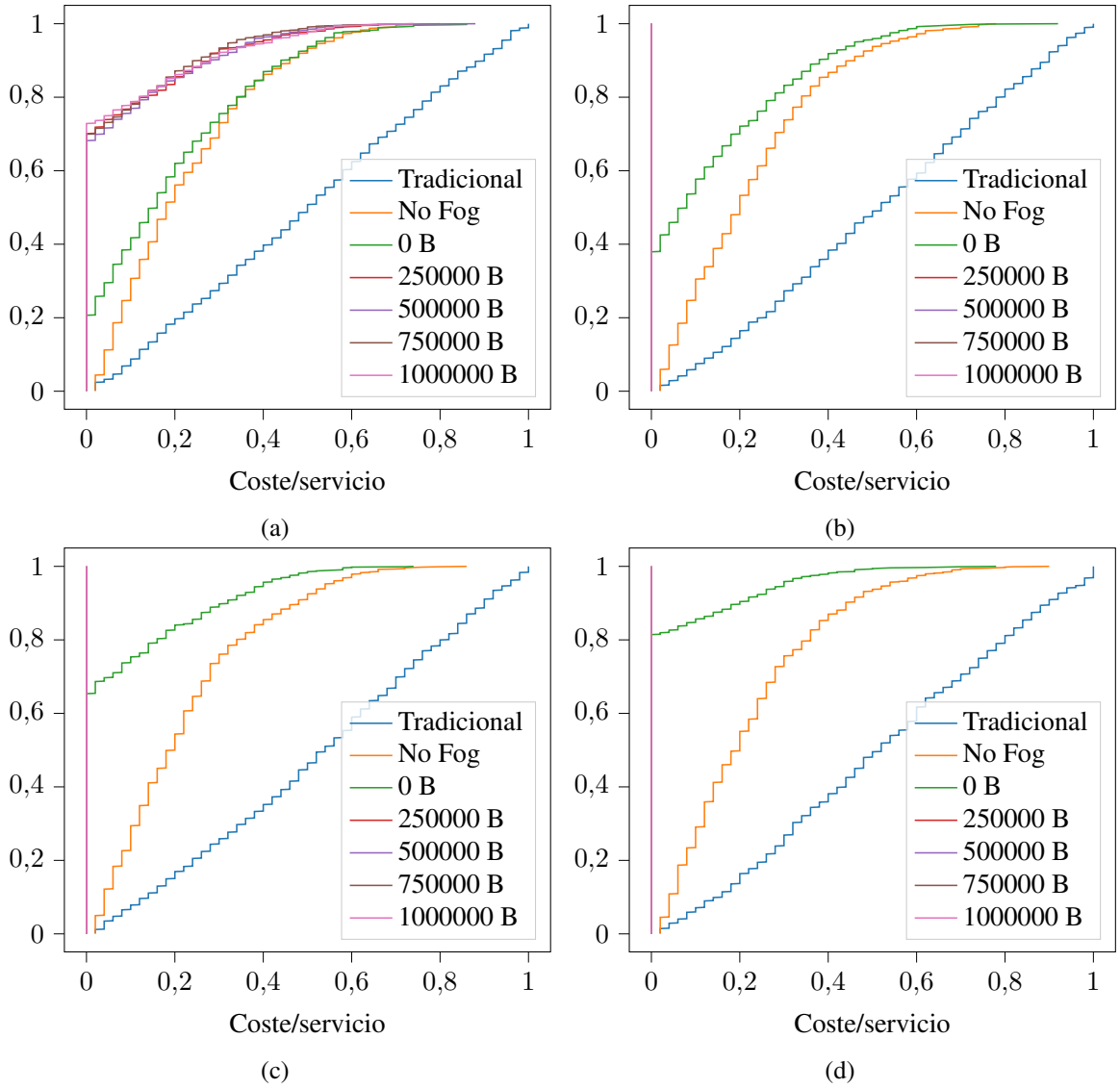


Figura 4.1: ECDF (coste/*servicio*). Tasa de generación: 10 pkts/s. Capacidad de procesado *Fog*: (a) 1000 Bps (b) 2000 Bps (c) 5000 Bps (d) 10000 Bps.

nodo *Fog*, el coste medio por *servicio* asociado a las mismas también presenta diferencias. Así, en la Figura 4.1a, donde, aunque la tasa de generación es bastante baja (10 pkts/s), es suficientemente elevada como para generar más bytes por segundo que los que el nodo *Fog* es capaz de procesar, el coste medio por *servicio* cuando se incluye el procesador se reduce en comparación a las dos configuraciones mencionadas previamente (donde toda la información se envía a los nodos *Cloud*). Además, si se incluye la cola de espera, la probabilidad de procesar los *servicios* localmente y, por lo tanto, a coste cero, aumenta considerablemente. Sin embargo, a medida que aumenta el tamaño de mencionada cola, el coste medio por *servicio* no se reduce, comportamiento que se puede asociar a una saturación de la cola debido a la limitada capacidad de procesado disponible.

Por otro lado, esta tendencia se replica en las simulaciones expuestas en las Figuras 4.1b, 4.1c y 4.1d, aunque con pequeñas diferencias. Primero, en la Figura 4.1b, la capacidad de procesado y la tasa de generación coinciden, es decir, se genera el mismo número de bytes por segundo que se puede procesar, aunque hay que tener en cuenta que el procesado se realiza a nivel de *servicio*. Esto resulta en un coste

medio por *servicio* menor en la tercera solución ('Cola 0 B') respecto a la misma en el escenario anterior, comportamiento razonable teniendo en cuenta que el procesador del nodo *Fog* es más rápido y, por lo tanto, más *servicios* son procesados con coste cero. Sin embargo, por el hecho de contar con una capacidad de procesado igual a la tasa de generación, se podría pensar que la totalidad de los *servicios* deberían de poder procesarse localmente. Esto no es así porque se está trabajando con tráfico de Poisson, por lo que todos los *servicios* no tienen el mismo número de paquetes, ni estos la misma longitud (aunque esto sea así en valor promedio). Debido a esto, en ciertas ocasiones llegarán *servicios* de mayor longitud que el procesador local no será capaz de terminar antes de que llegue el siguiente, el cual tendrá que ser enviado al nodo *Cloud* correspondiente ya que en esta solución no existe cola de espera. Por otro lado, en cuanto a las soluciones con una cola de espera en el nodo *Fog* incluidas en este escenario, todas presentan un coste medio por *servicio* nulo, lo que implica que todos los paquetes se procesan localmente. En estos casos, si algún *servicio* llega mientras el procesador está ocupado, este se podrá almacenar localmente, esperando a que el procesador se libere.

En cuanto a las Figuras 4.1c y 4.1d, donde la tasa de procesado local sigue aumentando, el comportamiento es exactamente el mismo al explicado en el escenario anterior. La única diferencia aparece en la solución sin cola de espera, la cual a medida que la capacidad de procesado aumenta, presenta un coste medio por *servicio* menor debido a que más *servicios* son procesados localmente a coste cero.

Observando estos resultados, está claro que incluir una política de costes para determinar dónde procesar la información reduce el coste considerablemente. Además, añadiendo capacidad de procesado y una cola de espera en los nodos *Fog*, el coste de procesar la información se reduce en mayor medida. Si se considerasen únicamente estos resultados, se puede concluir que una implementación *Fog-Cloud*, con una política de costes asociada, siempre presenta mejores resultados que la tradicional. Sin embargo, a parte del coste, es necesario considerar otros parámetros como el retardo, que se presenta a continuación. Por otro lado estos resultados permiten validar la implementación de la plataforma, que presenta un comportamiento predecible y valores de rendimiento lógicos ante gran variedad de configuraciones.

Considerando que la capacidad de procesado en los nodos *Cloud* es mucho mayor que en los nodos *Fog* en todos los escenarios expuestos, evitar enviar tráfico a los primeros con el objetivo de reducir el coste de procesado puede llegar a ser negativo en términos de retardo, siempre dependiendo de las características de la red disponible entre ambos tipos de nodos. La Figura 4.2 muestra el retardo medio que han sufrido todos los *servicios* procesados en los escenarios con una tasa de generación de 10 pkts/s y todas las tasas de procesado mencionadas.

En primer lugar, la aproximación 'tradicional' y la misma con la política de costes implementada ('No *Fog*') son las mejores soluciones en términos de retardo para las configuraciones elegidas. Además, el retardo medio total no varía entre ellas, ya que en todos los casos, el tiempo de espera y procesado es el mismo, el introducido por la red (que siempre es la misma), y el tiempo que se tardan en procesar todos los *servicios* en los nodos *Cloud* (los cuales tienen todos la misma capacidad). Así, para ambas soluciones, el tiempo de espera (desde que se genera el *servicio* hasta que este empieza a ser procesado) es de aproximadamente 600 milisegundos en todos los escenarios, mientras que el tiempo de procesado (los *servicios* tienen en media una longitud de 24000 Bytes con la configuración elegida en este caso), con una tasa de 100000 Bps en los nodos *Cloud*, ronda los 240 milisegundos.

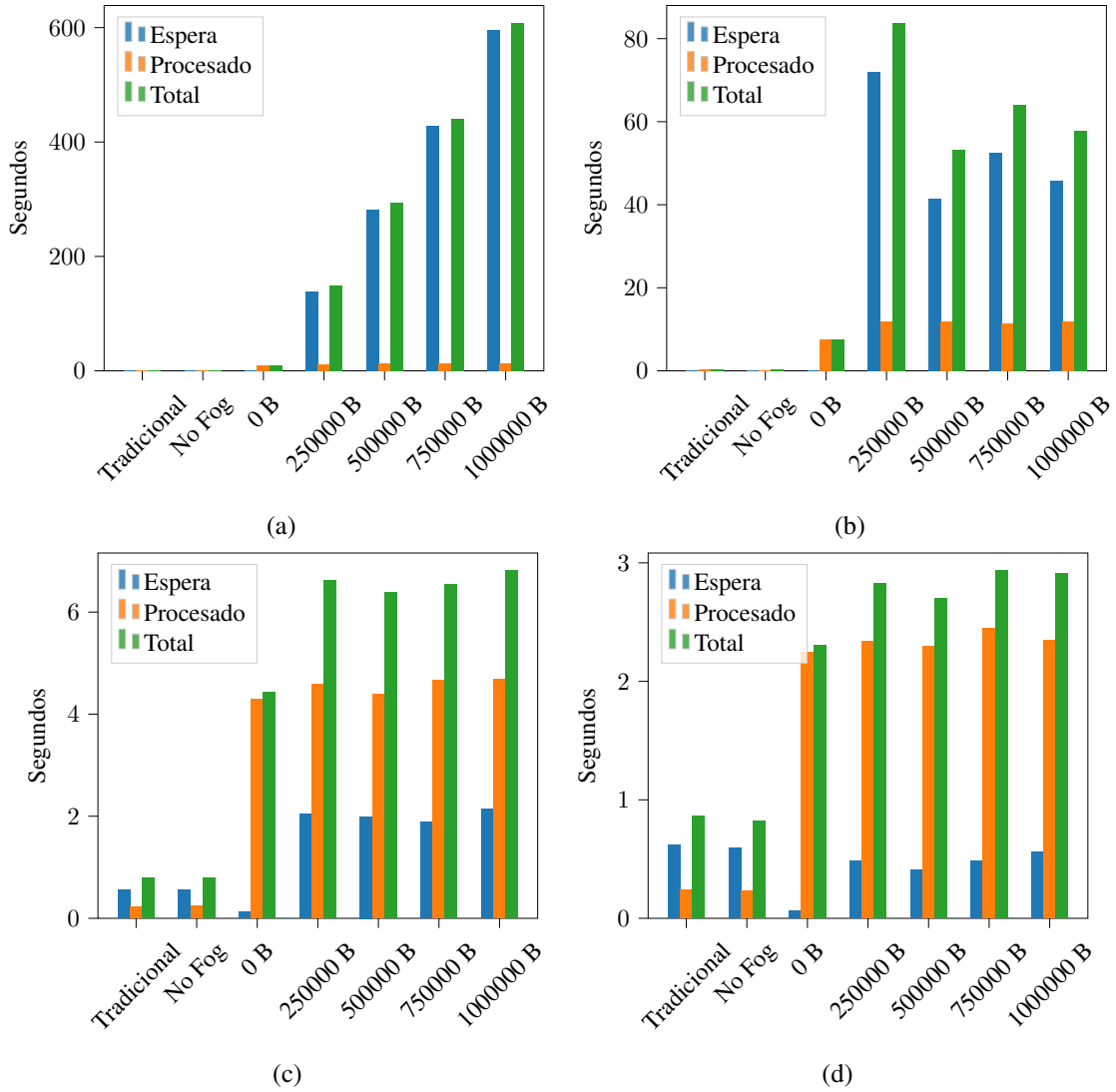


Figura 4.2: Retardo. Tasa de generación: 10 pkts/s. Capacidad de procesado *Fog*: (a) 1000 Bps (b) 2000 Bps (c) 5000 Bps (d) 10000 Bps.

Por otro lado, el resto de soluciones presentan diferencias en los distintos escenarios simulados. En cuanto a la Figura 4.2a, donde la capacidad de procesado del nodo *Fog* es de 1000 Bps (la menor de las cuatro), y por lo tanto es la que permite procesar menos paquetes localmente, es la que presenta peores resultados en términos de retardo para todas las soluciones expuestas. Observando esta gráfica, parece claro que añadir una cola de espera en el nodo local no es eficiente si la tasa de generación es superior a la de procesado, ya que esta se satura con rapidez e introduce un retardo continuo que no genera ningún beneficio, el cual es mayor a medida que aumenta el tamaño de mencionada cola.

Pasando a la 4.2b, donde la capacidad de procesado del nodo *Fog* es de 2000 Bps, que coincide con la tasa de generación de tráfico, en la solución sin cola de espera ('Cola 0 B') se reduce el retardo medio total con respecto al escenario anterior, siendo en este caso de aproximadamente 7 segundos y superando los 10 segundos en el caso anterior. En este caso, el principal motivo de este comportamiento es la mayor tasa de procesado local, que hace que los paquetes se procesen más rápido. Esto también implica que un número mayor de paquetes pueden ser procesados localmente, no atraviesan la red, y por lo tanto, se evita ese retardo adicional. Sin embargo, con la configuración elegida, el retardo introducido por la red en este

caso es prácticamente despreciable comparado con el tiempo de procesado. Para esta misma solución, también se puede observar cómo el tiempo medio de procesado es menor que en las cuatro soluciones que implementan una cola de espera local, comportamiento lógico considerando que la capacidad de procesado de los nodos *Cloud* es muy superior a la de los nodos *Fog*, y en las otras cuatro configuraciones los primeros no se utilizan.

En cuanto a las soluciones mencionadas, que añaden una cola de espera local, el tamaño de esta no es relevante en términos de retardo. Primero, en las cuatro soluciones expuestas se observa que el tiempo medio de procesado es el mismo. Este comportamiento se debe a que, en todos los casos, la totalidad de los *servicios* se ha procesado localmente, como se explicó con anterioridad sobre la Figura 4.1. En cuanto al tiempo de espera, las pequeñas variaciones que se aprecian entre las soluciones se deben a que el tráfico generado sigue el modelo de Poisson. Así, aunque en media todos los *servicios* y paquetes tienen la misma longitud, el orden de llegada de los paquetes de mayor o menor tamaño implican un tiempo de espera mayor o menor en la cola local, y por tanto, pequeñas variaciones en el retardo total. Sin embargo, si se comparan estas soluciones con las mismas presentes en el escenario anterior, se puede apreciar una clara reducción del retardo medio total, ya que, en este caso, la cola local no llega a saturarse en ningún momento.

Por último, las Figuras 4.2c y 4.2d presentan unos resultados muy similares. En la primera, con una capacidad de procesado local de 5000 Bps, se observa cómo el retardo medio se reduce considerablemente en todas las configuraciones si lo comparamos con los casos anteriores, situación que se repite en el último caso, donde la capacidad de procesado local aumenta hasta los 10000 Bps. Para las configuraciones con una cola de espera local, el hecho de incrementar la capacidad de procesado local, y considerando que ningún paquete se procesa de forma remota, explica estos resultados ya que, además de reducir el tiempo medio de procesado, también se reduce el tiempo medio de espera en la cola local. En cuanto a la configuración sin cola local, aunque se envíen más paquetes a los nodos *Cloud* en el primer caso y estos se procesen a una velocidad superior, en ambos escenarios la mayoría de los *servicios* son procesados localmente, donde la capacidad es superior en el segundo caso, que finalmente presenta un retardo total menor.

En resumen, incluir una política de costes al sistema reduce claramente el coste medio de procesado y no introduce ningún retardo adicional, por lo que es una opción que se debería implementar en la totalidad de los escenarios expuestos, ya que no presenta desventajas. Además, añadir capacidad de procesado en los nodos *Fog* reduce incluso más el coste medio de procesar los *servicios*, aunque aumenta el retardo respecto a las configuraciones tradicionales si la capacidad de procesado local no es suficientemente elevada. Por otro lado, añadir una cola de espera en los nodos *Fog*, si esta no tiene suficiente capacidad, no mejora el sistema, si no que lo empeora en términos de retardo ya que esta llega a saturar.

A continuación, se ha analizado el impacto sobre el comportamiento del sistema el aumento de tráfico de entrada. En concreto, la Figura 4.3 muestra la ECDF correspondiente a los escenarios con una tasa de generación de 100 pkts/s y todas las tasas de procesado mencionadas. En este caso, la tasa de generación de tráfico es 10 veces mayor que en los resultados expuestos anteriormente, y como se mantienen las misma capacidades de procesado, el número de bytes generados en todos los escenarios y configuraciones es muy superior al que los nodos *Fog* son capaces de procesar, por lo que en todos los casos, un gran número de *servicios* es enviado a los nodos *Cloud*.

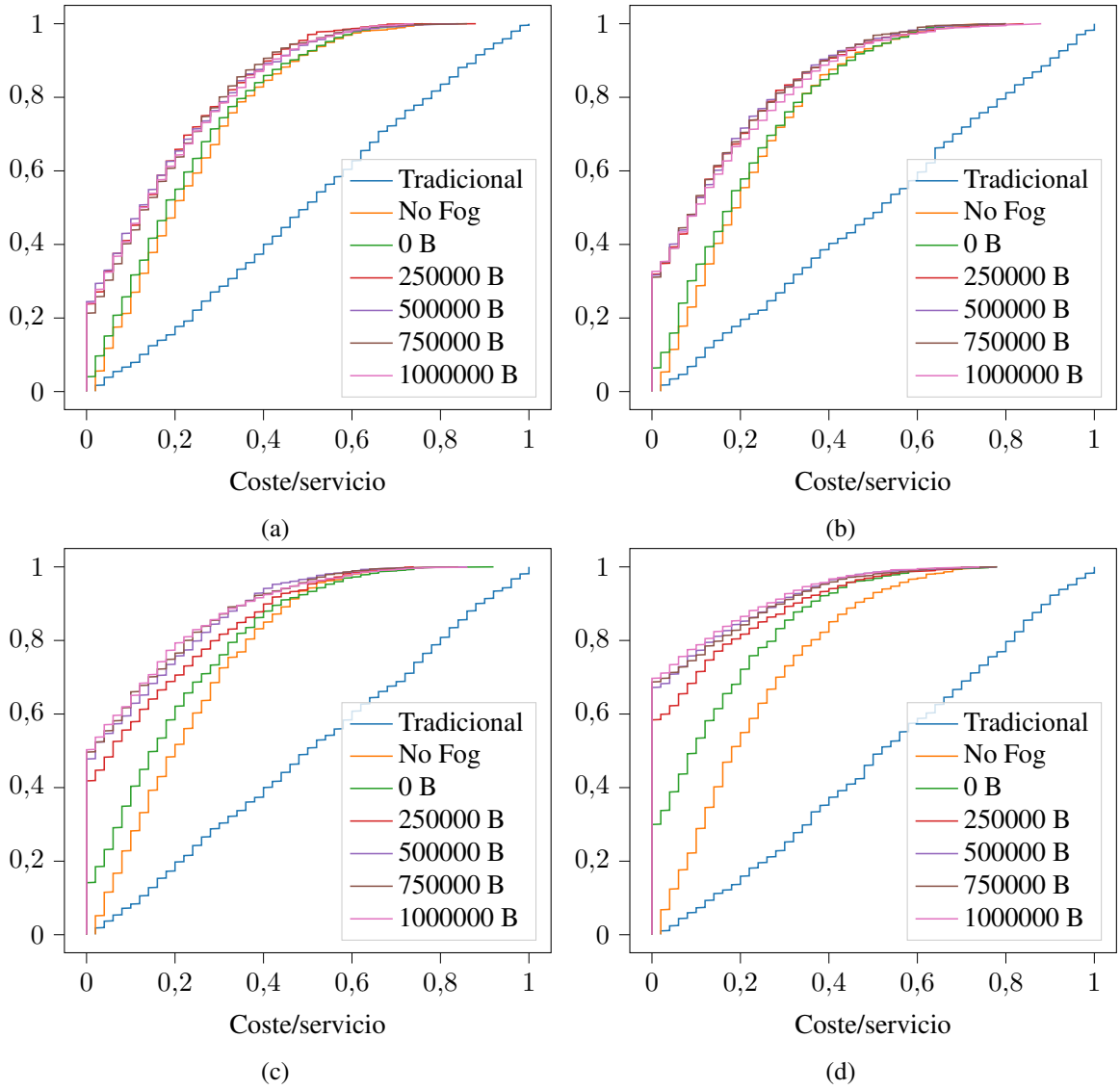


Figura 4.3: ECDF (coste/*servicio*). Tasa de generación: 100 pkts/s. Capacidad de procesado *Fog*: (a) 1000 Bps (b) 2000 Bps (c) 5000 Bps (d) 10000 Bps.

En primer lugar, observando las curvas correspondientes a las soluciones 'tradicional' y 'No *Fog*', el coste medio por *servicio* no varía respecto de las soluciones explicadas anteriormente, donde la tasa de generación era muy inferior. Este comportamiento es esperado, ya que de igual forma, todos los *servicios* se procesan remotamente, y el coste asociado a los nodos *Cloud* no se ha modificado. Sin embargo, el número total de *servicios* generados es superior (ya que el tiempo de simulación no ha cambiado), y aunque el coste medio por *servicio* no varíe, el hecho de procesar más *servicios* implica un coste total superior.

Por otro lado, comparando la configuración que añade capacidad de procesado en el nodo *Fog* pero no permite encolar ningún *servicio* ('Cola 0 B'), con la misma en los cuatro escenarios anteriores, el coste medio por *servicio* asociado es bastante superior. El hecho de procesar a la misma velocidad pero generar con una tasa superior implica que más paquetes necesitan ser enviados a los nodos *Cloud*, que tienen un coste superior. Además, un comportamiento que se apreciaba también en los cuatro casos explicados anteriormente entre las curvas 'No *Fog*' y 'Cola 0 B' se replica en estos escenarios. En la Figura 4.3a,

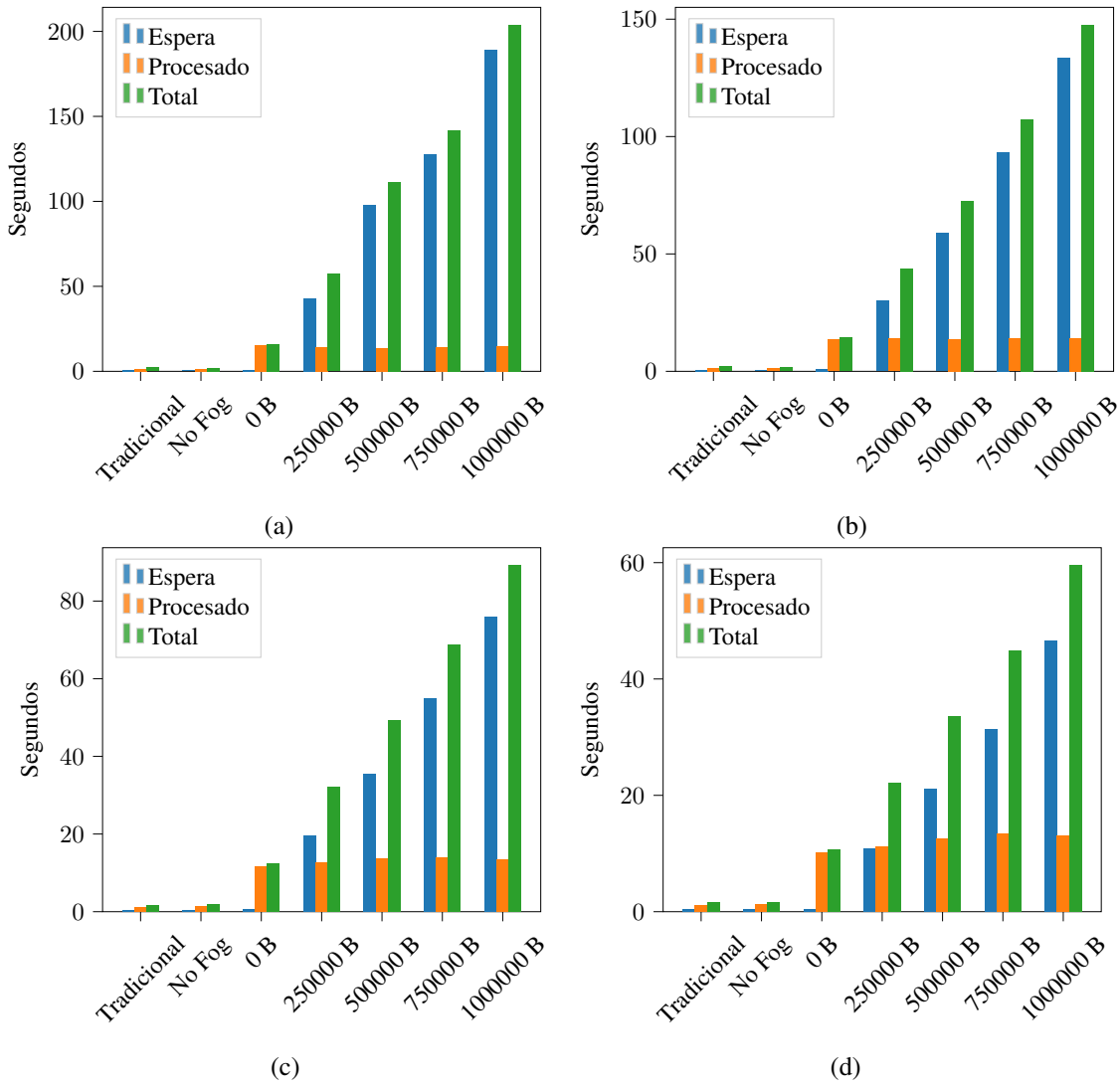


Figura 4.4: Retardo. Tasa de generación: 100 pkts/s. Capacidad de procesado *Fog*: (a) 1000 Bps (b) 2000 Bps (c) 5000 Bps (d) 10000 Bps.

donde la capacidad de procesado local es muy pequeña, esta prácticamente no influye en el coste medio por *servicio* final, ya que la mayor parte de los *servicios* se procesan remotamente. Sin embargo, a medida que se aumenta la capacidad de procesado del nodo *Fog* y por lo tanto más *servicios* se pueden procesar localmente, estas curvas se alejan, lo que implica que el coste medio por *servicio* de la solución con capacidad de procesado local se ve reducido (ver Figuras 4.3b, 4.3c y 4.3d).

En cuanto a las soluciones que incluyen una cola de espera local, estas permiten que un número aún mayor de *servicios* se procesen en el nodo *Fog*, por lo que el coste medio por *servicio* se reduce. Sin embargo, en la mayoría de escenarios, entre los distintos tamaños de cola no se aprecia ninguna diferencia, comportamiento que se debe a un tamaño de cola demasiado pequeño, y por lo tanto, que llega a saturar. Sin embargo, si se observa el último escenario (ver Figura 4.3d), el coste medio por *servicio* asociado a la curva con una cola de menor capacidad es superior al resto. En el momento que la tasa de procesado local aumenta, hay más posibilidades de que los *servicios* puedan procesarse sin pasar por la cola local (que aumentan incluso más cuanto mayor es su capacidad) y por lo tanto esta no llegue a saturar, lo que explica este comportamiento.

Por otro lado, la Figura 4.4 muestra el retardo medio de procesar los *servicio* para las mismas configuraciones y escenarios. En este caso, en los cuatro escenarios la tendencia se repite. Así, en la Figura 4.4a, donde la capacidad de procesado local es la más baja, el retardo medio total es el más alto, lo que se debe a dos motivos. Primero, a que los *servicios* que se procesen en el nodo *Fog* lo harán más lentamente. Segundo, esto implica que los que pasen por la cola local también van a incrementar el retardo medio total, ya que tienen que esperar a que el procesador local (siendo este el más lento) se quede libre. Por otro lado, es cierto que más *servicios* son enviados a los nodos remotos, y aunque todos tengan que atravesar la red, que incluye un retardo adicional, la capacidad de procesado de estos es muy superior (el tiempo de procesado es menor). Por ello, aunque pueda parecer que este escenario debería presentar los mejores resultados en términos de retardo, el tiempo de procesado y espera en la cola local son bastante elevados debido a la baja capacidad de procesado, por lo que condicionan los resultados medios finales.

A medida que aumenta la capacidad de procesado de los nodos *Fog*, el tiempo de procesado y espera en la cola local se reduce considerablemente, por lo que el retardo medio total disminuye hasta llegar a los valores mínimos presentes en estos cuatro escenarios (ver Figura 4.4d).

Finalmente, con todos los resultados expuestos hasta ahora parece claro que, tanto para una tasa de generación pequeña como para una superior, la inclusión de una política de costes en el sistema reduce el coste respecto a la solución más tradicional, y no introduce ningún retardo adicional. Sin embargo, en los escenarios expuestos en este documento, la inclusión de procesado en los nodos *Fog* no presenta una clara ventaja respecto a la solución tradicional. Aunque está claro que sí lo hace en términos de coste, ya que el asociado a los mismos es nulo, el retardo adicional que introduce su implementación respecto a las tradicionales podría no ser esperado, ya que una de las principales ventajas de las arquitecturas de tres capas IoT-Fog-Cloud mencionadas en la literatura es, precisamente, el ahorro temporal.

La tendencia observada en estos resultados se debe a las configuraciones implementadas, y depende principalmente de cuatro factores: la capacidad de procesado local, la longitud de la cola local, la capacidad de procesado remota y el retardo introducido por la red de comunicaciones. En todos los casos, tanto expuestos en este documento como en cualquier implementación en general, la capacidad de procesado de los nodos *Fog* es limitada, mientras que los nodos *Cloud* disponen de una muy superior. La relación entre ambas capacidades juega un papel fundamental a la hora de determinar la mejor solución en términos de retardo. Observando uno de los casos extremos, donde tanto nodos locales como remotos son capaces de procesar todos los *servicios* generados sin necesidad de utilizar colas de espera y obviando la red por ahora, parece obvio que el tiempo total de procesado cuando se utilizan los nodos *Cloud* siempre será menor que el mismo con los nodos *Fog*, ya que los primeros poseen una capacidad muy superior. Incluyendo la red de comunicaciones, al tiempo de procesado de los nodos *Cloud* habría que añadirle el tiempo adicional que introduce la red. Sin embargo, si las capacidades tanto de nodos locales como remotos no son suficientemente altas (como ocurre en todos los escenarios expuestos hasta ahora, donde la capacidad más grande se encuentra en los nodos *Cloud* y es, únicamente, de 100 KBps), el retardo adicional introducido por la red es despreciable, ya que el tiempo de procesado es muy superior, y por lo tanto, las configuraciones donde se mandan más paquetes a los nodos remotos presentan mejores resultados en este aspecto.

Para comprobar este comportamiento, a continuación se exponen los resultados obtenidos para escenarios con las mismas características expuestas en la Tabla 4.1, pero capacidades de procesado

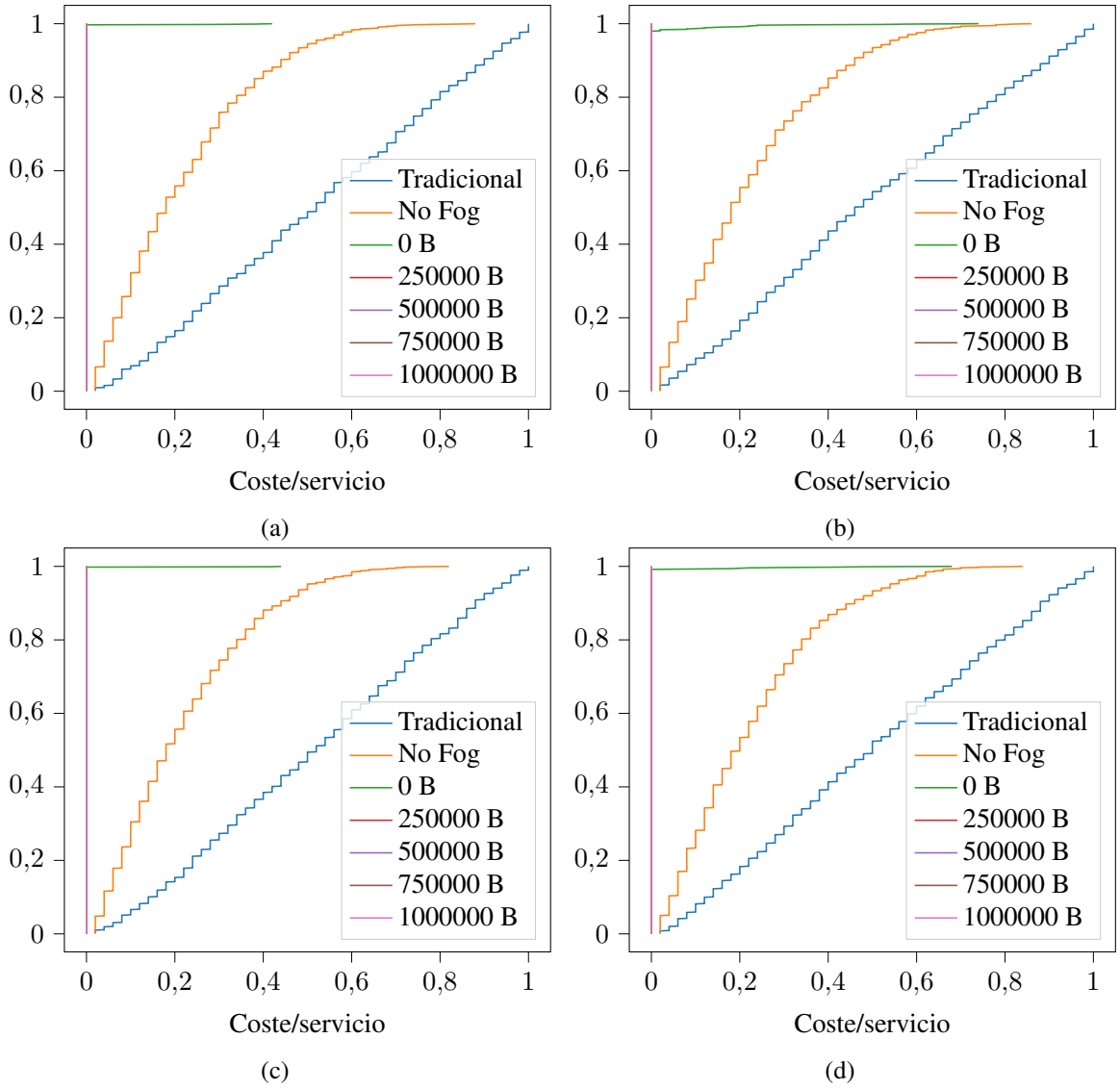


Figura 4.5: ECDF (coste/*servicio*). Capacidad de procesado *Fog* y Tasa de generación: (a) 0.5 MBps y 10 pkts/s (b) 0.5 MBps y 100 pkts/s (c) 1 MBps y 10 pkts/s (d) 1 MBps y 100 pkts/s.

superiores, tanto en los nodos *Cloud* (10 MBps), como en los nodos *Fog* (0.5 y 1 MBps).

En primer lugar, las gráficas expuestas en la Figura 4.5 muestran la ECDF para mencionadas configuraciones. En este caso, tanto la tasa de procesado local como remota se ven incrementadas con respecto a los escenarios expuestos anteriormente. Así, aunque la tasa de los nodos *Cloud* no tiene relevancia en estas gráficas, el hecho de aumentar la tasa de procesado en los nodos *Fog* implica que estos son capaces de procesar a mayor velocidad y, como la tasa de generación de tráfico no ha variado, un número mayor de *servicios* se procesan localmente. En relación al coste, las configuraciones 'tradicional' y 'No *Fog*' no varían, ya que todos los paquetes se procesan remotamente, siguiendo el mismo modelo de precios. Sin embargo, como se ha aumentado la tasa de procesado de los nodos *Fog*, con el resto de configuraciones se consiguen procesar la totalidad de los paquetes localmente, lo que implica un coste por *servicio* nulo. La única excepción se encuentra en la configuración sin cola local ('Cola 0 B'), donde todas las gráficas muestran que algún *servicio* se ha procesado remotamente. En el caso de las Figuras 4.5a y 4.5c (tasa de generación de 10 pkts/s) el número es tan pequeño que se podría despreciar, mientras que en las Figuras

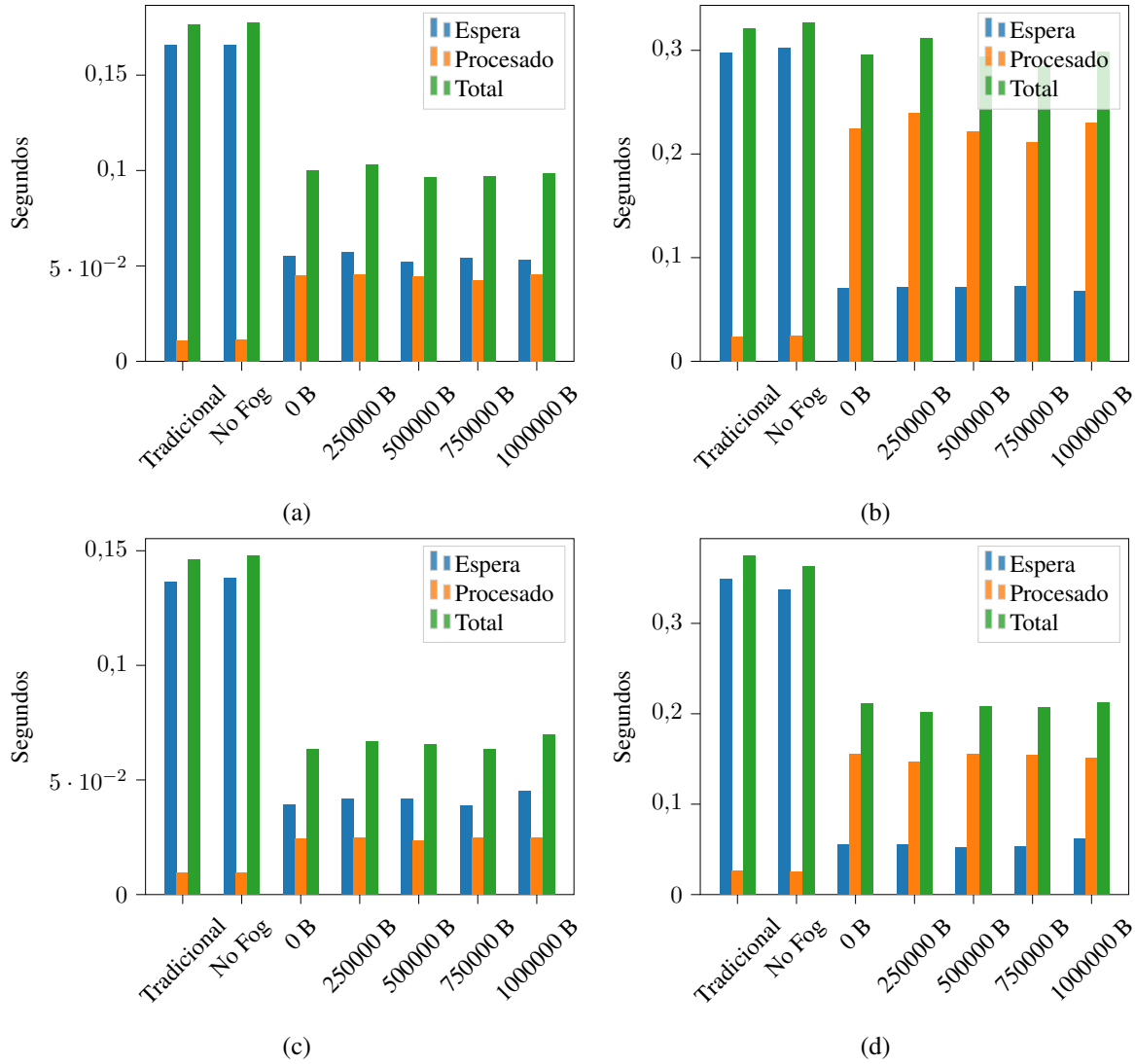


Figura 4.6: Retardo. Capacidad de procesado *Fog* y Tasa de generación: (a) 0.5 MBps y 10 pkts/s (b) 0.5 MBps y 100 pkts/s (c) 1 MBps y 10 pkts/s (d) 1 MBps y 100 pkts/s.

4.5b y 4.5d (tasa de generación de 100 pkts/s), aunque de igual forma la mayoría de los paquetes son procesados localmente, alguno más se ha tenido que enviar a los nodos *Cloud*.

Por otro lado, la Figura 4.6 muestra el retardo que presentan estos escenarios. En este caso, tanto la tasa de procesado de los nodos *Fog* como la de los nodos *Cloud* tiene mucha relevancia. Primero, el hecho de aumentar ambas respecto a las mismas en los escenarios anteriormente expuestos, implica que el tiempo medio de procesado se reduce considerablemente. Como se explicó con anterioridad, trabajar con tasas de procesado muy pequeñas implica que el tiempo de procesado es bastante elevado, muy superior al retardo que introduce la red, por lo que este último es despreciable. Por ello, en los escenarios previos, una de las principales ventajas de la computación *Fog* mencionadas en la literatura (retardos inferiores) no se apreciaba, observando un comportamiento contrario. Sin embargo, cuando la capacidad de procesado de los nodos *Fog* se incrementa, enviar *servicios* para procesar remotamente puede no ser tan beneficioso en términos de retardo, ya que, aunque los paquetes se procesen más rápido en los nodos *Cloud*, el retardo introducido por la red es muy superior al tiempo ahorrado en el procesado.

Así, comparando las Figuras 4.6a y 4.6c (tasa de generación de 10 pkts/s) y 4.6b y 4.6d (tasa de generación de 100 pkts/s), en ambos casos el tiempo medio de procesado en las configuraciones que incluyen nodos *Fog* es menor cuanto mayor es la capacidad de procesado en estos. Por último, en los cuatro escenarios, las configuraciones con procesado local incluido presentan unos resultados prácticamente iguales en términos de retardo. Esto se debe a que prácticamente todos los paquetes pueden ser procesados sin pasar por la cola de espera, por lo que el hecho de aumentar el tamaño de esta no implica ninguna modificación.

En conclusión, como se esperaba, el hecho de incluir una tasa de procesado local superior implica siempre mejores resultados, tanto en términos de coste (ya que se pueden procesar más *servicios* localmente) como en términos de retardo (se procesan más rápido). Sin embargo, aumentar la capacidad de procesado local también implica un coste adicional. Por ello, a la hora de implementar configuraciones de este tipo, es conveniente comprobar las capacidades necesarias para obtener unos resultados aceptables, sopesando la inversión inicial que implica utilizar nodos *Fog* más potentes, respecto al ahorro y las prestaciones que estos proporcionan en el sistema.

Así, el desarrollo de esta plataforma se plantea como una solución sencilla, rápida y realista para comprobar este tipo de configuraciones, consiguiendo resultados relevantes sin necesidad de realizar despliegues en entornos reales, ahorrando así todo el coste asociado a los mismos.

5 | Conclusiones

Finalmente, una vez expuesta la plataforma desarrollada, su funcionamiento, y las funcionalidades que proporciona, además de presentar un conjunto de resultados comparando las configuraciones tradicionales de la computación *Cloud*, con las nuevas aproximaciones que incluyen soluciones Fog-Cloud combinadas, este capítulo expone las conclusiones finales que se obtienen a partir de todo lo expuesto anteriormente.

Además, como esta plataforma se ha creado como una solución abierta a la cual se pueden añadir funcionalidades adicionales según surjan nuevos requisitos, este capítulo también recopila una serie de posibles líneas futuras para continuar su desarrollo.

5.1. Conclusiones

En primer lugar, la plataforma desarrollada se ha implementado utilizando la tecnología Docker y Docker *Compose* para crear la red de nodos, y el lenguaje de programación Python para implementar la lógica de estos. Como se ha mencionado en secciones anteriores, estas herramientas dotan a la plataforma de características como ligereza, rapidez en la creación y destrucción de despliegues o la facilidad de añadir nuevas funcionalidades a la misma, entre otras. Así, todos los resultados que se han obtenido durante el desarrollo de este trabajo y expuesto en este documento, a parte de permitir realizar una comparación entre despliegues tradicionales y soluciones combinadas, cuyas conclusiones se comentan a continuación, han servido para validar el correcto funcionamiento de la plataforma y comprobar que todas las características mencionadas por las que se ha elegido la tecnología utilizada, están presentes en la misma.

Así, a parte de las mencionadas comprobaciones, los resultados obtenidos tienen como objetivo principal exponer las ventajas y desventajas asociadas a las soluciones combinadas IoT-Fog-Cloud respecto a las soluciones tradicionales, donde solo se utiliza computación en la nube. Para ello, el Capítulo 4 recoge los resultados obtenidos en todos los escenarios simulados, exponiendo y explicando las diferencias encontradas, principalmente en términos de coste y retardo.

Una de las primeras comparaciones llevadas a cabo ha consistido en realizar un despliegue completamente tradicional utilizando computación en la nube, es decir, procesando toda la información obtenida de los dispositivos IoT en un mismo nodo *Cloud* predeterminado. Esta estrategia se ha comparado con una política de decisión basada en el coste, es decir, entre varios nodos *Cloud* disponibles, escoger el más barato en el momento de enviar la información. Los resultados obtenidos en este caso son bastante claros. En primer lugar, considerando que todos los nodos *Cloud* utilizados disponen de las mismas

características, el hecho de implementar la política de decisión basada en el precio actual de los mismos, reduce el coste medio de procesar la información en todos escenarios planteados. Este comportamiento es lógico ya que la primera solución planteada solo dispone de un nodo al cual enviar la información, siendo el coste computacional final el valor medio que presenta ese. Sin embargo, la segunda solución dispone de varios nodos que, aunque individualmente presenten el mismo coste medio que el anterior, decidir en cada momento cuál es el más barato permite reducir el coste final considerablemente.

Comparando estas mismas soluciones en términos de retardo, se observa cómo los resultados obtenidos coinciden ya que, aunque la segunda solución tenga más nodos *Cloud* a su disposición, estos tienen las mismas características (igual capacidad de procesado, almacenamiento, etc.). Por ello, se puede concluir que, incluso antes de incorporar computación *Fog* a un sistema, el hecho de implementar una política de decisión basada en costes cuando existen varios nodos *Cloud* disponibles siempre supone un ahorro, además de no empeorar otras características como puede ser el retardo.

La siguiente comparativa realizada implica añadir computación *Fog*, es decir, dotar de capacidad de computo y almacenamiento a los nodos *Fog*, pudiendo así procesar parte de la información obtenida en los nodos IoT de forma local. Para ello, la primera solución comparada con las dos mencionadas previamente ha consistido en añadir diferentes tasas de procesado local (sin incluir colas de espera) y procesar la mayor cantidad de información posible de forma local (ya que estos nodos tienen asociado un coste nulo). Así, en términos de coste los resultados son muy claros. Cuando la capacidad de procesado local es muy pequeña, y por lo tanto la mayoría de los paquetes tienen que ser enviados a los nodos *Cloud*, el coste medio de procesar la información es muy similar al obtenido con la solución anterior (con la política de costes implementada). Por otro lado, a medida que la capacidad de procesado local aumenta y, por lo tanto, más *servicios* pueden ser procesados en los nodos *Fog*, el coste medio por *servicio* comienza a reducirse, hasta el punto en el que este vale cero, situación dada cuando la capacidad local es tan elevada que todos los *servicios* generados se pueden procesar localmente. Por lo tanto, dependiendo del coste adicional que suponga dotar de capacidad de procesado a los nodos *Fog*, si esta es suficiente, el coste medio de procesado siempre se verá reducido y, en el peor de los casos, presentará los mismos resultados que cuando no se incluye, pero nunca peores en términos de coste.

Sin embargo, en términos de retardo los resultados no son favorables para esta solución en todos los escenarios. En los primeros que se han expuesto, donde las capacidades de procesado de los nodos, tanto locales como remotos, no son muy elevadas, el hecho de procesar paquetes localmente implica un aumento del retardo total, ya que el tiempo de procesado de los nodos *Fog* es siempre mayor que el de los *Cloud*. Al contrario de lo expuesto en la literatura, en estos casos las soluciones tradicionales son más rápidas que las combinadas, por lo que una de las principales ventajas de la computación *Fog* no se cumple. Esto se debe al hecho de contar con capacidades de procesado muy pequeñas, donde el retardo introducido por la red de comunicaciones es despreciable. Así, a medida que las capacidades de procesado aumentan (aunque se siga manteniendo una capacidad superior en los nodos *Cloud*), el retardo medio cuando se usan soluciones combinadas comienza a presentar mejores resultados que las soluciones tradicionales, ya el retardo introducido por la red (el cual se evita con la computación *Fog*) que antes era despreciable, pasa a ser el cuello de botella en términos de retardo.

Los últimos escenarios incluidos y comparados han sido los que, además de incluir capacidad de procesado local, disponen de una cola de espera para almacenar algún *servicio* cuando el procesador está

ocupado. En términos de coste, para todos los escenarios planteados, el hecho de añadir la cola de espera implica que más paquetes pueden ser procesados localmente, por lo que, el coste medio de procesado se reduce, aunque cuando esta es muy pequeña y la generación de tráfico muy alta, el número adicional es muy reducido, y el ahorro es prácticamente despreciable. Así, se puede concluir que, a medida que la cola local aumenta, el coste medio total se reduce y tanto el tamaño de la misma como la tasa de llegadas de nuevos *servicios*, determinan si lo hace mucho, o la reducción es prácticamente despreciable.

Por otro lado, en términos de retardo, el hecho de añadir una cola de procesado local presenta diferentes resultados dependiendo de las configuraciones elegidas. Primero, cuando la capacidad de procesado local es muy pequeña y la mayoría de los *servicios* generados se procesan remotamente, la cola local se satura rápidamente, añadiendo un retardo constante a todos los paquetes procesado en los nodos *Fog* que aumenta cuanto mayor es el tamaño de esta, y que además, no reduce el coste medio total. Por ello, cuando la capacidad de procesado de los nodos locales es muy inferior a la tasa de generación o llegada de *servicios*, incluir una cola de procesado, a parte de no reducir el coste, incluye un retardo adicional constante debido a la saturación de la misma, por lo que no es una implementación aconsejable.

Sin embargo, en configuraciones donde la tasa de procesado es suficientemente elevada como para que el procesador local esté disponible la mayoría del tiempo, añadir una cola como apoyo adicional previendo la llegada de algún *servicio* muy pesado o varios muy seguidos que impliquen un procesador ocupado en la siguiente llegada, permiten almacenar estos *servicios* sin necesidad de incrementar el coste debido al uso de computación en la nube, sin añadir un retardo relevante ya que la cola no se satura.

Finalmente, a modo de resumen, parece claro que añadir una política de decisión basada en el precio a las soluciones tradicionales permite reducir considerablemente el coste medio de procesar la información generada por los dispositivos IoT. Además, cuando se añade capacidad de procesado a los nodos *Fog*, el precio se reduce aún más, presentando mejores resultados cuanto mayor es esta capacidad. Sin embargo, si esta no es suficientemente elevada como para que el tiempo de procesado sea inferior al retardo introducido por la red, utilizar soluciones con computación *Fog* puede implicar un retardo superior, situación que se invierte cuando la red pasa a ser el cuello de botella. Por último, cuando se añade una cola de espera local, es importante realizar los cálculos correctos para que esta no sature, ya que de ser así, el coste medio de procesar la información no mejorará en gran medida, mientras que el retardo introducido por esta será muy relevante.

5.2. Líneas futuras

Por último, como se ha mencionado previamente en el documento, la plataforma desarrollada no es una solución cerrada, por lo que, la finalización de este trabajo no limita la implementación de funcionalidades adicionales en un futuro, cuando surjan nuevos requisitos que no se puedan satisfacer con lo desarrollado hasta ahora o cuando se quieran analizar diferentes estrategias. Además, a parte de los resultados obtenidos en este documento, se pueden simular otros escenarios con diferentes características y políticas que puedan ser relevantes en distintas aplicaciones o sectores. Así, a continuación se exponen algunas ideas que pueden ser interesantes en un futuro.

Primero, en cuanto a la plataforma, hay ciertas funcionalidades no implementadas que pueden ser

de gran utilidad. Con lo desarrollado hasta ahora, los escenarios que se pueden plantear incorporan un único nodo *Fog* y tantos nodos *Cloud* como se considere. Un primer paso puede ser la incorporación de una función que permita utilizar tantos nodos *Fog* como se necesiten, pudiendo establecer en cada uno condiciones de red particulares.

Asimismo, la versión actual implementa un modelo de *servicio* genérico, pero que podría no ser adecuado para ciertas aplicaciones. En el futuro se contempla definir diferentes modelados de *servicio* que incluyan la posibilidad de procesado paralelo, procesado en varias etapas (*pipeline*), *servicios* con prioridades o requisitos estrictos de retardo.

Respecto a las capacidades de cómputo, también se contempla añadir capacidad de multi-procesador a los nodos, tanto *Fog* como *Cloud*, lo que aumentaría la complejidad de las decisiones.

Por otro lado, al inicio del trabajo, se planteó la posibilidad de utilizar el orquestador de contenedores Kubernetes en la plataforma. Esta idea se descartó, ya que con la primera versión de la misma, se pretendían realizar despliegues sencillos en una única máquina física, por lo que no se necesitaba. Con la posibilidad de realizar despliegues de mayor tamaño (número mayor de nodos) y distribuidos en diferentes máquinas, incorporar este u otro orquestador de contenedores puede ser una opción a tener en cuenta.

Finalmente, en cuanto a todos los resultados expuestos en este documento, se ha utilizado una política de decisión basada en el coste, es decir, enviando la información al nodo más barato disponible. Sin embargo, no todas las aplicaciones o servicios priorizan este modelo, por lo que, implementando otras políticas de decisión, basadas en retardo, seguridad, localización, estado de las colas, o incluso otro modelo de precios distinto al utilizado, se pueden obtener resultados con otras características más adecuadas a otro tipo de escenarios.

A | Dockerfile - Imagen de nodos *Fog*

```
FROM ubuntu:18.04

LABEL mantainer='David'

RUN apt-get update && apt-get -y install python python3-pip \
    net-tools iputils-ping vim && apt-get update && \
    apt-get -y install dh-autoreconf sudo apache2 dnsmasq \
    dnsmasq-base dnsutils apache2-bin debhelper && \
    apt -y install protobuf-compiler apache2-dev libprotobuf-dev \
    git libxcb-present-dev autotools-dev dh-autoreconf pkg-config \
    libcairo2-dev && apt -y install libpango1.0-dev libssl-dev \
    ssl-cert iptables iproute2 && pip3 install numpy

RUN git clone https://gitlab.tlmat.unican.es/ldiez/mahimahi-mod.git \
    && cd mahimahi-mod && ./autogen.sh && \
    ./configure && make && make install

RUN useradd user1 -m -s /bin/bash

RUN echo "user1:user1" | chpasswd

RUN usermod -aG sudo user1

COPY . /home/user1

CMD ["/home/user1/mmPython.py"]

ENTRYPOINT ["python3"]
```

B | Dockerfile - Imagen de nodos *Cloud*

```
FROM ubuntu:18.04

LABEL mantainer='David'

RUN apt-get update && apt-get -y install python python3-pip \
    net-tools iputils-ping vim && apt-get update && \
    apt-get -y install dh-autoreconf sudo apache2 dnsmasq \
    dnsmasq-base dnsutils apache2-bin debhelper && \
    apt -y install protobuf-compiler apache2-dev libprotobuf-dev \
    git libxcb-present-dev autotools-dev dh-autoreconf pkg-config \
    libcairo2-dev && apt -y install libpango1.0-dev libssl-dev \
    ssl-cert iptables iproute2 && pip3 install numpy

COPY . /home

CMD ["/home/cloudRec.py"]

ENTRYPOINT ["python3"]
```

C | Dockerfile - Imagen de nodos *Master*

```
FROM ubuntu:18.04

LABEL mantainer='David'

RUN apt-get update && apt-get -y install python python3-pip \
    net-tools iputils-ping vim && apt-get update && \
    apt-get -y install dh-autoreconf sudo apache2 dnsmasq \
    dnsmasq-base dnsutils apache2-bin debhelper && \
    apt -y install protobuf-compiler apache2-dev libprotobuf-dev \
    git libxcb-present-dev autotools-dev dh-autoreconf pkg-config \
    libcairo2-dev && apt -y install libpango1.0-dev libssl-dev \
    ssl-cert iptables iproute2 && pip3 install numpy

COPY . /home/

CMD ["/home/master.py"]

ENTRYPOINT ["python3"]
```

D | Fichero Docker-Compose - Ejemplo

```
version: '3'
services:
########################## Nodo Master ##########################
  master_node:
    image: master_nodes
    privileged: true
    volumes:
      - ./Volumes/Master:/home/logs
    environment:
      - N=50
      - SIM_TIME=60
      - FOG_NODES=1
      - CLOUD_NODES=4
      - MOD_TYPE=FPM
      - DIS_TYPE=ONE
      - FOG_COST=0
      - TRADICIONAL=yes

########################## Nodos Cloud ##########################
  #Nodo Cloud 1
  cloud_node_1:
    image: cloud_nodes
    privileged: true
    volumes:
      - ./Volumes/Cloud1:/home/logs
    environment:
      - CAP_PRO=1000 #Bytes/s
      - SIM_TIME=60 #segundos
      - FOG_NODES=1

  #Nodo Cloud 2
  ...
```

```
  #Nodo Cloud 3
  ...

  #Nodo Cloud 4
  ...

########################## Nodos Fog ##########################
  #Nodo Fog 1
  fog_node_1:
    image: fog_nodes
    privileged: true
    volumes:
      - ./Volumes/Fog:/home/user1/logs
    depends_on:
      - master_node
      - cloud_node_1
      - cloud_node_2
      - cloud_node_3
      - cloud_node_4
    environment:
      - TRACE_UP=/mahimahi/traces/TMobile-LTE-short.up
      - TRACE_DOWN=/mahimahi/traces/TMobile-LTE-short.down
      - CAP_PRO=2000 #Bytes/s
      - PKT_LEN=200 #Bytes
      - TR_R=10 #Pkts/s
      - SERV_R=0.083 #Serv/s
      - CONT_NAME=fog_node_1
      - TR_DIS=POISSON
      - SIM_TIME=60
      - CLOUD_NODES=4
      - COLA=0
```

Bibliografía

[1] P. Suresh, J. V. Daniel, V. Parthasarathy, and R. H. Aswathy, "A state of the art review on the internet of things (iot) history, technology and fields of deployment," in *2014 International Conference on Science Engineering and Management Research (ICSEMR)*, pp. 1–8, 2014.

[2] P. Gokhale, O. Bhat, and S. Bhat, "Introduction to iot," *International Advanced Research Journal in Science, Engineering and Technology*, vol. 5, pp. 41–44, 2018.

[3] H. Boyes, B. Hallaq, J. Cunningham, and T. Watson, "The industrial internet of things (iiot): An analysis framework," *Computers in Industry*, vol. 101, pp. 1–12, 2018.

[4] K. Shafique, B. A. Khawaja, F. Sabir, S. Qazi, and M. Mustaqim, "Internet of things (iot) for next-generation smart systems: A review of current challenges, future trends and prospects for emerging 5g-iot scenarios," *IEEE Access*, vol. 8, pp. 23022–23040, 2020.

[5] S. Dahmen-Lhuissier, "Etsi - mobile technologies - 5g, 5g specs | future technology." (accedido: 13 de Julio, 2021).

[6] H. Tyagi and R. Kumar, "Cloud computing for iot," in *Intenet of Things (IoT)*, pp. 25–41, 2020.

[7] M. Mohammed Sadeeq, N. M. Abdulkareem, S. R. M. Zeebaree, D. Mikaeel Ahmed, A. Saifullah Sami, and R. R. Zebari, "Iot and cloud computing issues, challenges and opportunities: A review," *Qubahan Academic Journal*, vol. 1, no. 2, p. 1–7, 2021.

[8] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, p. 13–16, Association for Computing Machinery, 2012.

[9] M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[10] G. Peralta, M. Iglesias-Urkia, M. Barcelo, R. Gomez, A. Moran, and J. Bilbao, "Fog computing based efficient iot scheme for the industry 4.0," *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, pp. 1–6, 2017.

[11] X. Masip-Bruin, E. Marín-Tordera, G. Tashakor, A. Jukan, and G.-J. Ren, "Foggy clouds and cloudy fogs: a real need for coordinated management of fog-to-cloud computing systems," *IEEE Wireless Communications*, vol. 23, no. 5, pp. 120–128, 2016.

[12] H. Gupta, A. Vahid Dastjerdi, S. K. Ghosh, and R. Buyya, "ifogsim: A toolkit for modeling and simulation of resource management techniques in the internet of things, edge and fog computing environments," *Software: Practice and Experience*, vol. 47, no. 9, p. 1275–1296, 2017.

[13] A. Kertesz, T. Pflanzner, and T. Gyimothy, "A mobile iot device simulator for iot-fog-cloud systems," *Journal of Grid Computing*, vol. 17, no. 3, p. 529–551, 2019.

[14] I. Lera, C. Guerrero, and C. Juiz, "Yafs: A simulator for iot scenarios in fog computing," *IEEE Access*, vol. 7, pp. 91745–91758, 2019.

[15] A. M. Potdar, N. D G, S. Kengond, and M. M. Mulla, "Performance evaluation of docker container and virtual machine," *Procedia Computer Science*, vol. 171, pp. 1419–1428, 2020. Third International Conference on Computing and Network Communications (CoCoNet'19).

[16] Y. Li, A.-C. Orgerie, I. Rodero, B. L. Amersho, M. Parashar, and J.-M. Menaud, "End-to-end energy models for edge cloud-based iot platforms: Application to data stream analysis in iot," *Future Generation Computer Systems*, vol. 87, pp. 667–678, 2018.

[17] C. Powell, C. Desiniotis, and B. Dezfouli, "The fog development kit: A platform for the development and management of fog systems," *IEEE Internet of Things Journal*, vol. 7, no. 4, pp. 3198–3213, 2020.

[18] Z. Benomar, D. Bruneo, F. Longo, G. Merlino, and A. Puliafito, "A mininet-based emulated testbed for the i/ocloud," in *2019 15th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, pp. 277–283, 2019.

[19] R. Netravali, A. Sivaraman, S. Das, A. Goyal, K. Winstein, J. Mickens, and H. Balakrishnan, "Mahimahi: Accurate record-and-replay for HTTP," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pp. 417–429, USENIX Association, 2015.

[20] G. Peralta, P. Garrido, J. Bilbao, R. Agüero, and P. M. Crespo, "On the combination of multi-cloud and network coding for cost-efficient storage in industrial applications," *Sensors*, vol. 19, no. 7, 2019.